[redacted]

Knowledge Systems Laboratory
Report No. KSL 86-38

August 1986

A Layered Environment for Reasoning about Action

by

Barbara Hayes-Roth, Alan Garvey, M. Vaughan Johnson Jr., and Michael Hewett

(NASA-CR-190703) A LAYERED ENVIRONMENT FOR REASONING ABOUT ACTION (Stanford Univ.) 129 p

N92-71085

Unclas

Z9/82 0115490

KNOWLEDGE SYSTEMS LABORATORY
Computer Science Department
Stanford University
Stanford, California 94305

A Layered Environment for Reasoning about Action[1]

Barbara Hayes-Roth, Alan Garvey, M. Vaughan Johnson Jr., and Micheal Hewett
Stanford University

August, 1986

DRAFTKSL-86-38***

[1]The research was supported by the following grants: NIH Grant #RR-00785; NIH Grant #RR-00711; NSF Grant #DMB84-2038; NASA/Ames Grant #NCC 2-274; Boeing Grant #W266875; DARPA Contract #N00039-83-C-0136. We thank the following individuals for their contributions. Jeff Harvey, Bob Schulman, and Craig Cornelius worked on BB1. Russ Altman translated the PROTEAN knowledge sources from their original implementation in BB1 to their current implementation in BB1-ACCORD. Bill Pardee and Paul Rosenbloom made useful suggestions on presentation. Frederick Hayes-Roth provided constructive criticism at several stages of the work. We especially thank Bruce Buchanan and Ed Feigenbaum for sponsoring the work within the Knowledge Systems Laboratory.

Table of Contents

Abstract

An intelligent system reasons about--controls, explains, learns about--its actions, thereby improving its efforts to achieve goals and function in its environment. In order to perform effectively, a system must have knowledge of the actions it can perform, of the events and states that can occur, and of the relationships among particular instances of those actions, events, and states. We propose to represent such knowledge in a hierarchy of knowledge abstractions and to impose uniform standards of knowledge content and representation on modules within each hierarchical level. At the root of this hierarchy, the BB1 *blackboard control architecture* can support a variety of task-specific *frameworks*, each of which can accommodate a range of domain-specific *applications*. Conversely, an application system such as PROTEAN instantiates the knowledge structures in a framework such as ACCORD, which instantiates the knowledge structures in BB1. We refer to the evolving set of such modules as the *BB* environment*. To illustrate our approach, we describe: (a) BB1 and its capabilities for control, explanation, and learning; (b) ACCORD, a framework for solving *arrangement problems* by means of an *assembly method*; (c) two applications of BB1-ACCORD, the PROTEAN system for protein-structure modeling and the SIGHTPLAN system for designing construction-site layouts; and (d) two hypothetical *multi-faceted systems* that integrate ACCORD and PROTEAN with other frameworks and applications. We show how ACCORD enhances PROTEAN's use of BB1's reasoning capabilities and how it facilitates the building of new application systems, such as SIGHTPLAN and the two multi-faceted systems. We summarize the current state of the BB* environment and our plans for extending it. Finally, we assess BB* both as a computing environment and as a theory of intelligent systems.

1. Overview

"Human intelligence depends essentially on the fact that we can represent in language facts about our situation, our goals, and the effects of the various actions we can perform." John McCarthy [35]

"In the knowledge is the power." Edward A. Feigenbaum [14]

"The fact, then, that many complex systems have a nearly decomposable, hierarchic structure is a major facilitating factor enabling us to understand, to describe, and even to 'see' such systems and their parts." Herbert A. Simon [47]

We begin with a premise: *An intelligent system reasons about its actions.* Of course, we do not mean to suggest that a system should engage in extended contemplation of every one of its computational and physical actions, but rather: (a) that it can reason about many of its actions; (b) that it does reason about them much of the time; and (c) that its reasoning improves its efforts to achieve goals and otherwise function in its environment.

A system might reason about its actions in various ways and with various consequences (see Figure 1a) . For example, a system might *control* its actions: decide which actions to perform at particular points in time. Control reasoning can affect the resources the system consumes in pursuing a goal, the side effects it produces, and the probability of achieving its goal [8, 9, 13, 17, 23, 26, 27]. As a second example, a system might *explain* its actions: describe the ways in which the actions it intends to perform or has performed serve its goals. Explanation typically serves social functions, such as teaching another individual how to perform a task or persuading another individual that one is performing the task competently [5, 6, 21, 22]. As a third example, a system might *learn* about its actions: modify its ability or inclination to perform particular actions in appropriate circumstances. Learning enables the system to expand and improve its capabilities [24, 31, 32, 35, 38, 39, 45]. While a system could perform many other important types of reasoning about its actions, we focus on control, explanation, and learning.

Insert Figure 1

Given the premise above, we put forth a hypothesis: *In order to perform effectively, an intelligent system must have knowledge of its actions.* It must have knowledge of the actions it can perform, of the events and states that can occur, and of the relationships among particular instances of these actions, events, and states. For example, it must know: the actions that are relevant to its current task; the enabling conditions required by particular actions; the cost, reliability, and side effects of particular actions; the internal and external events and states whose occurrences contribute to or hinder performance of its task; the power of particular actions to bring about particular events and states; and the power of external forces to bring about particular events and states.

In our work, we formulate explicit, interpretable representations of these and other kinds of knowledge (see Figure 1b) as a foundation for intelligent behavior. Thus, we define "knowledge" broadly, as "that which is known."[2] In fact, most computational objects in our systems (all except the basic architectural cycle, low-level data-retrieval functions, and user interface) appear as elements of a well-structured, modular, declarative knowledge base. As such, they are amenable to knowledge-level operations, such as acquistion, modification, verification, deduction, induction, instantiation, and comparison. Moreover, we can incrementally improve almost any aspect of a system's behavior by extending the depth or extent of its knowledge. We have begun to construct an expanding edifice of such knowledge for a variety of *problem classes, problem-solving methods*, and *subject-matter domains*.

In constructing this edifice, we emphasize a design principle: *We represent knowledge in an abstraction hierarchy*. Although "true" knowledge abstractions probably lie on a continuum, we currently focus on three particular levels--architecture, framework, and application.

At the most general level, we define an *architecture* to comprise: (a) the set of basic knowledge structures used to represent all actions, events, states, and facts in a system; and (b) a mechanism for instantiating, choosing, and executing actions. Architectural knowledge is independent of problem class, problem-solving method, and subject-matter domain. For example, the *blackboard control architecture* [23], which is implemented as the BB1 system discussed below, supports applications as varied as protein-structure analysis [4, 25, 29], process planning [41], and autonomous vehicle control [43]. In addition, BB1 provides specific knowledge structures and a powerful mechanism to support intelligent control, explanation, learning.

At the intermediate level, we define a *framework* as the set of knowledge structures used to represent actions, events, states, and facts involved in performing a particular *task*. That is, a framework comprises the knowledge structures involved in solving a particular class of

[2]The American Heritage Dictionary of the English Language, 1981, "Knowledge," definition # 3.

problems with a particular method, but independent of subject-matter domain. For example, the *arrangement-assembly framework*, which is implemented as the ACCORD knowledge base discussed below, embodies the knowledge used to solve *arrangement problems* by means of an *assembly method*. However, the knowledge in ACCORD applies to arrangement-assembly tasks in such varied subject-matter domains as protein-structure analysis, construction-site layout, and travel planning.

At the most specific level, we define an *application* as the set of knowledge structures that instantiate particular actions, events, states, and facts to solve a particular class of problems by means of a particular method in a particular subject-matter domain. For example, the PROTEAN system [4, 25, 29] embodies the knowledge used to determine the three-dimensional structures of proteins--that is, to solve arrangement problems in the domain of protein chemistry by means of the assembly method.

As illustrated in Figure 1c (see also Table 1), BB1, ACCORD, and PROTEAN are elements of a knowledge abstraction hierarchy. BB1 can accommodate a variety of modular frameworks, one of which is ACCORD. Similarly, ACCORD (and each other framework) can accommodate a range of modular applications, one of which is PROTEAN. (As Figure 1c shows, many current applications are implemented directly in BB1.) We refer to the evolving set of such modules as the *BB* environment*.

Conversely, a given application system composes modules from the BB* environment in several layers of implementation. For example, PROTEAN's knowledge about constructing proteins instantiates and configures a number of ACCORD's more general knowledge structures for assembling arrangements. Similarly, ACCORD's knowledge structures instantiate and configure a number of BB1's still more general knowledge structures about problem-solving, control, explanation, and learning. When PROTEAN goes to work on a problem, its actions are interpreted through these several layers of implementation.

Insert Table 1

In adapting this widely accepted software engineering principle--generally referred to as *modular and layered design* [18, 19, 49]--to intelligent systems, we achieve several advantages. First, each abstraction level offers certain representational and computational services to higher levels, while shielding them from the details of implementation. Second, we can understand complex systems in terms of their simpler modular components. Third, we can investigate and test alternative implementations of modules at one level independently of the modules at other levels. Fourth, we can eliminate levels from applications that do not require their services. Fifth, we can achieve additive and, in some cases, multiplicative improvements in efficiency across levels [44]. Finally, we can apply general knowledge modules in an appropriate variety of contexts and configure selected lower-level knowledge modules for a variety of specific purposes.

We impose one additional constraint on our knowledge abstraction hierarchy: *Modules within a level must meet uniform standards of knowledge content and representation.* Accordingly, we adopt a single architecture, BB1. Although BB1 accommodates multiple frameworks, each of them must provide the same core categories of knowledge within a specified representation scheme. Similarly, each application must provide another set of core knowledge categories within another specified representation scheme.

This constraint offers several related advantages. First, we can define new application systems by configuring and augmenting existing knowledge modules within a level. Second, we can identify and eliminate redundancy in the contents of independently acquired modules within an application system. Third, we can organize modules in any appropriate organizational scheme. In particular, we can organize them in a conventional "pipeline," such that a succession of modules receive, process, and pass on information. Alternatively, we can organize them to operate more intimately: operating simultaneously, sharing intermediate results, and affecting one another's behavior. In fact, a system can reason about how to select and organize modules to solve new problems. Fourth, we can superimpose generic capabilities for control, explanation, and learning upon the designated configurations of modules. In sum, uniformity of content and representation within a level allows us to achieve the conventional capabilitiy of *open systems interconnection* [51] and to strive toward a more ambitious capability that we will call *open systems integration.* It raises the possibility of incrementally increasing the quantity and variety of knowledge within an application system, while preserving a well-structured foundation and a coherent face for the system as a whole (see Figure 1d).

Our objectives in this work are two-fold. First, we wish to develop a rich and varied family of reusable modules for building intelligent systems. System builders should be able to build new systems by configuring appropriate subsets of these modules in appropriate organizational schemes. Where new modules are needed, system builders should be able to introduce them into the existing family and integrate them into new systems with ease. The resulting systems should be well-structured, perspicuous, modifiable, and extensible. Second, we wish to develop a theory of intelligent systems. The theory must provide: (a) a great range of problem-solving skills, including the ability to solve a variety of problem classes with a variety of problem-solving methods in a variety of subject-matter domains; (b) the ability to apply any available knowledge to improve problem-solving performance; and (c) the ability to reason about--control, explain, and learn about--action. We believe that our approach to developing the BB* environment enables us to progress toward both objectives.

The remainder of this paper develops and substantiates the four themes introduced above and displayed in Figure 1 as follows. Section 2 provides an overview of BB1 and presents examples from the PROTEAN system to illustrate BB1's architectural support for general problem solving, as well as for intelligent control, explanation, and learning. Section 3 presents the arrangement-assembly framework (ACCORD) to illustrate the standard knowledge content and representation required to define a framework. It also describes the associated BB1 framework-interpreter. Section 4 illustrates reasoning within the BB* environment. It shows how PROTEAN is reimplemented in ACCORD, describes the advantages of that implementation and illustrates its enhancement of PROTEAN's reasoning abilities. Section 5 discusses knowledge engineering within the BB* environment. It describes the design and implementation of a prototype SIGHTPLAN system [50] (for designing construction-site layouts) within BB1-ACCORD and examines the applicability of ACCORD to arrangement-assembly tasks in other domains. Section 5 also introduces a new class of *multi-faceted systems* to illustrate BB*'s capability for open systems integration. Section 6 discusses the current state of the BB* environment and our plans for extending it. Section 7 highlights the major results of the paper.

2. BB1: An Architecture for Control, Explanation, and Learning

2.1 Overview of BB1

BB1 (see Figure 2) provides a uniform blackboard architecture for systems that reason about their own actions as well as about particular problems and solutions. In a BB1 system, functionally independent *knowledge sources* cooperate to solve problems by recording and modifying solution elements in a global data structure called the *blackboard*. A system may have three classes of knowledge sources. *Domain knowledge sources* solve domain problems on a *domain blackboard* and send and receive messages along input/output channels. *Control knowledge sources* construct control plans for the system's own behavior on *control blackboard*. *Learning knowledge sources* modify knowledge sources and facts in the system's *knowledge base*. All knowledge sources operate simultaneously and, when triggered, compete for scheduling priority. BB1 also provides an explanation capability by which a system shows how its actions fit into its control plan. The explanation module currently operates as part of the basic execution cycle. On each cycle, the user has an opportunity to request an explanation.

The BB1 execution cycle comprises three steps:

1. The *interpreter* executes the action of one knowledge source, making changes to the contents of the appropriate blackboard or the knowledge base.

2. The blackboard changes satisfy the conditions of other domain, control, and learning knowledge sources. The *agenda-manager* adds corresponding *KSARs (knowledge source activation records)* to the *agenda*.

3. The *scheduler* rates each KSAR on the agenda against the current control plan and, using a *scheduling rule* that is recorded on the control blackboard, chooses one KSAR to execute its action. Unless it has been instructed to operate autonomously, the scheduler also invites the user to request an explanation for the chosen action or any of several other kinds of information or to override the scheduler's chosen action with another one.

Insert Figure 2

Building an application system in BB1 entails: (a) building domain knowledge sources to reason about a problem and its solution on a domain blackboard; and (b) building control knowledge sources to reason about problem-solving strategy on the BB1 control blackboard. It may also entail building a knowledge base of information used by these knowledge sources. To support these activities, BB1 provides generic knowledge structures for user-specified blackboards and knowledge sources and a tool called BBEdit for instantiating them. In addition, it provides knowledge structures for: blackboard events, KSARs, agendas, the control blackboard and its levels. These knowledge structures are instantiated internally by BB1 during program execution. However, they also may be accessed by user-specified knowledge sources. BB1 provides a variety of access functions for operating on instances of all of these knowledge structures, as well as on intermediate results generated by knowledge sources during problem solving.

Since we have discussed BB1's knowledge structures and processes in detail elsewhere [23], we do not repeat that material here. Instead, we use the PROTEAN system, which is described briefly in the next section, to illustrate BB1's knowledge structures and problem-solving style and, especially, to demonstrate BB1's capabilities for intelligent control, explanation, and learning.

2.2 An Illustrative BB1 Application: PROTEAN

2.2.1 PROTEAN's Problem: Protein-Structure Analysis

PROTEAN's task is to identify the three-dimensional conformations of proteins.[3]

Its input data include information about a test protein's primary and secondary structures. The test protein's *primary structure* is its defining sequence of amino acids. For example, Figure 3 shows the primary structure of a protein called the lac-repressor headpiece. In addition the atomic architecture of each individual amino acid is known. For example, Figure 4 shows the architectures of two amino acids, alanine and tyrosine. The protein's *secondary structure* is the sequence of higher-order sub-units (*alpha-helixes, beta-sheets,* and random-coils) defined by the pattern of turns in the protein's primary structure. For example, Figure 3 shows the secondary structure of the lac-repressor headpiece.

---

[3]The PROTEAN project is directed by Bruce Buchanan and Oleg Jardetzky. The research team includes: Barbara Hayes-Roth, Russ Altman, Jim Brinkley, John Brugge, Craig Cornelius, Bruce Duncan, Alan Garvey, and Olivier Lichtarge.

Insert Figure 3

Insert Figure 4

The input data also include a number of constraints on the test protein's conformation (see Table 2). For example, there may be about 50-60 *NOEs (Nuclear Overhauser Effects)*, each of which indicates that two particular atoms in the protein are within 3-10 angstroms of one another. There may be evidence that certain atoms are accessible to solvent, indicating that they lie near the molecular surface of the protein. There may be information about the overall size, shape, and density of the protein molecule.

Insert Table 2

Based on these different kinds of knowledge, PROTEAN must identify the test protein's *tertiary structure*--the folding of its primary and secondary structures in three-dimensional space (see Figure 5). Because the problem is underconstrained, there may be many conformations that satisfy the available constraints. PROTEAN must identify the entire family of such conformations. Moreover, since proteins are known to be mobile in solution, PROTEAN must reason about potential mobility in the conformations it identifies.

Insert Figure 5

2.2.2 PROTEAN's Approach: The Assembly Method

PROTEAN is designed to analyze protein structure by means of the following *assembly method*.

Because protein-structure analysis entails a large combinatoric search space, PROTEAN reasons about a test protein's conformation at four different levels of abstraction (see Figure 6): *molecule, solid, superatom,* and *atom*. By reasoning at a high level of abstraction first, PROTEAN can reduce the number of conformations it must consider at a lower level of abstraction. Conversely, it can use the details of reasoning at a low level of abstraction to restrict the family of conformations it accepts at a higher level of abstraction.

Insert Figure 6

PROTEAN's basic problem-solving operation is to apply one or more constraints between two protein structures, determining where one structure can lie given: (a) its current hypothesized position; (b) its constraints with the other structure; and (c) the current hypothesized position of the other structure. PROTEAN currently uses a generate-and-test procedure to sample space at some level of resolution and to identify all *locations* in which a structure satisfies a given set of constraints. Figure 7a schematizes PROTEAN's application of constraints to position one helix relative to another helix whose position is fixed. Figure 7b schematizes PROTEAN's application of constraints to restrict further the locations previously identified for two helices.

Insert Figure 7

PROTEAN applies constraints in the context of one or more *partial arrangements (pa)*. Each partial arrangement *includes* a subset of the structures in a test protein and a subset of the constraints among those structures. PROTEAN designates one structure as the *anchor*, thereby declaring that it has an arbitrary, fixed location. It positions all other structures in the partial arrangement relative to the anchor. When PROTEAN applies constraints between a structure and the anchor, we say that it *anchors* an *anchoree* to the anchor. When it applies constraints between a structure that has no constraints with the anchor and some anchoree, we say that it *appends* an *appendage*. Finally, when PROTEAN applies constraints between two anchorees or appendages to reduce both of their locations, we say that it *yokes* them.

Insert Figure 8

Because protein analysis entails a combinatoric search, PROTEAN must control its search intelligently. It must reason about: how to group protein structures in partial solutions; which structure to designate the anchor of each partial solution; when to perform anchoring, appending, and yoking operations to apply particular constraints between particular objects;

when to refine partial solutions at different levels of abstraction; and when to combine overlapping partial solutions. This reasoning must incorporate general computational principles, such as: choosing an anchor that has many constraints to many other structures; focusing on structures that have been restricted to small families; and preferring constraints that maximally restrict a structure's family. It must also incorporate biochemistry knowledge such as: defining the space of potentially useful constraints; and characterizing the constraining power of different constraints.

2.3 Domain Reasoning in BB1

2.3.1 Overview of Domain Reasoning

As indicated above, BB1 systems solve problems through the actions of domain knowledge sources that contribute solution elements to a shared representation on the blackboard. BB1 provides generic knowledge structures for instantiation as user-specified blackboards and knowledge sources. Let us consider PROTEAN's solution blackboard and domain knowledge sources.

2.3.2 Illustrative Domain Knowledge Sources

PROTEAN's solution blackboard has the four levels of abstraction defined above: molecule, solid, superatom, and atom. Each level specifies a frame that may be instantiated as particular objects at that level. For example, Figure 9 shows an object at the solid level, helix1.

Insert Figure 9

PROTEAN's current knowledge sources perform the grouping, anchoring, yoking, and appending actions describe above. Each one instantiates a generic knowledge source frame with these attributes:

- The *trigger* specifies a set of event-based predicates. When all trigger predicates are true, a knowledge source is triggered.

- The *context* specifies variable-value bindings that distinguish different contexts in which to perform the action of a triggered knowledge source. A separate KSAR is created for each such context and placed on the agenda.

- The *precondition* specifies a set of state-based predicates. The action of a triggered knowledge source can be executed only when all precondition predicates are true.

- The *obviation condition* specifies a set of state-based predicates. If at any time, all of a KSAR's obviation conditions are true, it is removed from the agenda of pending KSARs.

- The *KS vars* specify local variable bindings for the knowledge source action. (The knowledge source also can specify local variable bindings within each other attribute.)

- The *KS action* is a set of rules that, when instantiated and executed, add or modify information on some blackboard or in the knowledge base.

- Other attributes (e.g., *cost, reliability*) specify other characteristics of the knowledge source.

Figure 10 shows an illustrative PROTEAN knowledge source, Yoke-Structures. Figure 10 also illustrates some of the BB1 blackboard access *$functions* available to system builders. Figure 11 shows an illustrative Yoke-Structures KSAR. Figure 12 shows an illustrative blackboard event produced by executing the KSAR shown in Figure 11.

Insert Figure 10

Insert Figure 11

Insert Figure 12

2.3.3 Varieties of Domain Reasoning in BB1

Like other blackboard systems, BB1 accommodates a variety of inference mechanisms within the domain reasoning component of a single application system. For example, different PROTEAN knowledge sources may: reason top-down to hypothesize protein conformations at the Superatom level that are compatible with hypothesized conformations at the solid level; reason bottom-up to hypothesize conformations at the solid level that are compatible with hypotheses at the superatom level; reason within the solid level to hypothesize positions for secondary structures that are compatible with one another; reason from first principles with its generate-and-test mechanism; or reason from knowledge by inserting known partial arrangements into the hypothesized conformation.

2.4 Control Reasoning in BB1

2.4.1 Overview of Control

As indicated above, BB1 provides a uniform blackboard architecture for reasoning about domain and control problem-solving.

Control knowledge sources incrementally construct a dynamic control plan to guide the system's behavior. The plan itself is a loosely hierarchical description of classes of actions that are desirable during different problem-solving time intervals. Since different control knowledge sources can embody qualitatively different sorts of knowledge, a BB1 system can reason about control by means of top-down, bottom-up, or opportunistic methods. They may include application-specific knowledge sources defined by the user, along with various generic control knowledge sources provided by BB1.

When executed, control knowledge sources add or modify objects on BB1's architecturally defined control blackboard. In general, decisions at the *strategy* level prescribe sequences of subordinate strategies or foci. Decisions at the *focus* level represent related collections of heuristics against which KSARs are rated. Decisions at the *heuristic* level specify individual functions for rating KSARs.

Like domain knowledge sources, control knowledge sources are triggered by changes to the blackboard and add KSARs to the agenda where they compete with all other KSARs for scheduling priority. As a result, a BB1 system incrementally constructs and modifies a dynamic control plan throughout its problem-solving activities. Depending upon the knowledge available, the system may systematically elaborate a high-level strategy or reason in a more opportunistic fashion. The BB1 scheduler simply chooses KSARs that satisfy the current control plan.

The following trace from PROTEAN illustrates the basic elements of BB1's approach to control reasoning.

2.4.2 An Illustrative Trace of Control Reasoning

In the following trace, PROTEAN incrementally elaborates the simple control plan shown in Figure 13.

Insert Figure 13

The trace begins when a user asks PROTEAN to analyze the lac-repressor headpiece. This request triggers the knowledge source Post-the-Problem. Since the resulting KSAR is the only one on the agenda at this time, the scheduler chooses it, creating the Problem description shown in Figure 14.

Insert Figure 14

The new problem triggers a control knowledge source, Develop-PS-of-Best-Anchor, and a domain knowledge source, Post-Solid-Anchors. Because PROTEAN has not yet formulated a control plan, the scheduler uses the default scheduling rule (posted by convention at the heuristic level of the control blackboard), which favors control actions. Develop-PS-of-Best-Anchor generates the corresponding strategy decision, which appears in Figure 13 and, in more detail, in Figure 15. Two of the strategy's attributes, *procedure-type* and *procedure-data*, specify that PROTEAN desires to perform a sequence of two kinds of actions, first those that create the best anchor-space and then those that position all secondary structures within that anchor-space.

Insert Figure 15

The new strategy triggers BB1's generic control knowledge source Initialize-Prescription, which is shown in Figure 16. Using the strategy's procedure-type and procedure-data attributes, it determines that "Create-Best-Anchor-Space" should be the new strategy's Current-Prescription.

Insert Figure 16

The new strategy also triggers BB1's generic control knowledge source, Terminate-Decision. However, Terminate-Decision specifies a precondition that will not be satisfied until the strategy's *goal* predicates (see Figure 15) are true. Only then will this Terminate-Decision KSAR appear on the agenda of executable KSARs.

The new Current-Prescription triggers the control knowledge source Create-Best-Anchor-Space. It creates the corresponding focus decision, which appears in Figure 13 and, in detail, in Figure 17.

Insert Figure 17

The new focus triggers four control knowledge sources, one for each of the heuristics it names: Prefer-Activate-Anchor-Space, Prefer-Rigid-Anchors, Prefer-Long-Anchors, and Prefer-Constraining-Anchors. Each one creates a corresponding heuristic decision, as shown in Figure 13. As illustrated by Prefer-Activate-Anchor-Space, which appears in Figure 18, each heuristic specifies a *function* with which to evaluate KSARs.

Insert Figure 18

The new focus decision also triggers BB1's generic control knowledge source, Terminate-Decision. However, Terminate-Decision specifies a precondition that will not be satisfied until the new focus decision's goal predicates (see Figure 17) are true. Only then will this Terminate-Decision KSAR appear on the agenda of executable KSARs.

Now PROTEAN has executed all pending control knowledge sources and elaborated a complete control plan for the first phase of its strategy for the lac-repressor headpiece. At this point, there is only one KSAR on the agenda, the one involving the domain knowledge source Post-Solid-Anchors, which was triggered by the original posting of the lac-repressor problem above. The scheduler chooses it, creating one object at the solid level representing each of the seven secondary structures in the lac-repressor headpiece (see, for example, the representation of helix1 in Figure 9).

Each new solid triggers the domain knowledge source Activate-Anchor-Space. Thus, there are now seven KSARs on the agenda. PROTEAN uses the heuristics in its current focus to rate the alternative KSARs. For example, Figure 19 shows that KSAR15 has high ratings against most of the current heuristics. As a consequence, the scheduler chooses KSAR15, whose action creates an anchor space with helix1 as the anchor.

Insert Figure 19

Creation of the anchor space for helix1 triggers the domain knowledge source, Add-Anchoree-to-Anchor-Space. It generates six KSARs, one for each of the other secondary structures in the protein.

Since creation of the anchor space for helix1 also renders the current focus decision's goal predicates true, it satisfies the precondition of the Terminate-Decision KSAR that was triggered by that focus. The scheduler chooses that Terminate-Decision KSAR. Its action changes the *status* of the focus and each of its subordinate heuristics to "inoperative."

The change in the focus decision's status triggers BB1's generic control knowledge source Update-Prescription and it is chosen by the scheduler. Using the strategy's procedure-type and procedure-data (see Figure 15), Update-Prescription determines that Create-Best-Anchor-Space is now the strategy's *expired-prescription*, while Position-All-Structures is its new Current-Prescription.

The new Current-Prescription triggers the control knowledge source Position-All-Structures. The scheduler chooses it and its action creates a new focus, Develop-PS-of-Helix1, which appears in Figure 13 and, in detail, in Figure 20.

Insert Figure 20

The new focus triggers seven control knowledge sources, one for each of the heuristics it names: Prefer-Strategically-Selected-Anchor, Prefer-Rigid-Anchorees, Prefer-Long-Anchorees, Prefer-Constraining-Anchorees, Prefer-Constrained-Anchorees, Prefer-Strong-Constraints, Prefer-Anchoring-over-Yoking. The scheduler chooses to execute each one, thereby creating corresponding heuristics.

The new focus also triggers BB1's generic control knowledge source, Terminate-Decision. Again, Terminate-Decision specifies a precondition that will not be satisfied until the new focus decision's goal predicates are true. Only then will this Terminate-Decision KSAR appear on the agenda of executable KSARs.

Now PROTEAN has executed all pending control knowledge sources and elaborated a complete control plan for the second phase of its strategy for the lac-repressor headpiece. At this point, there are six pending KSARs on the agenda, each of which applies the domain knowledge source Add-Anchoree-to-Anchor-Space to a different secondary structure in the protein. As illustrated in Figure 21, PROTEAN now rates each KSAR against the new heuristics associated with the current focus, but not against any of the heuristics associated with the previous focus. The scheduler chooses the highest-priority KSAR, KSAR34, whose action adds helix2 as an anchoree in helix1's anchor space.

Insert Figure 21

The new anchoree triggers the domain knowledge source Anchor-Helix, which adds several KSARs to the agenda, one for each constraint between helix2 and helix1. Each of these new KSARs is rated against the current control heuristics and competes with the pending KSARs for scheduling priority. The scheduler chooses the highest-priority KSAR, KSAR30, which adds helix3 as an anchoree in helix1's anchor space.

The new anchoree triggers the domain knowledge source Anchor-Helix again, which adds KSARs to the agenda for each constraint between helix3 and helix1. Each of these new KSARs is rated against the current control heuristics and competes with the pending KSARs for scheduling priority. The scheduler chooses the highest-priority KSAR, KSAR33, which anchors helix2 to helix1 with the constraint NOE1.

In the remainder of the trace, which we omit for brevity, PROTEAN adds the remaining secondary structures as anchorees in helix1's anchor space and applies constraints among all of the structures with appropriate anchoring, yoking, and appending operations. The agenda of pending KSARs grows considerably longer as newly added structures create opportunities for anchoring and as newly anchored structures create opportunities for yoking and appending. The BB1 scheduler continues to choose these different actions opportunistically, based on their combined ratings against PROTEAN's current control heuristics. Eventually, PROTEAN positions all structures relative to helix1, thereby satisfying the second focus decision's goal. As a consequence, the associated Terminate-Decision KSAR becomes executable and the scheduler chooses it. Terminate-Decision changes the status of the second focus and each of its heuristics to "inoperative." These events satisfy the top-level strategy's goal and its associated Terminate-Decision becomes executable. The scheduler chooses it and its action changes the strategy's status to "inoperative." PROTEAN thereby completes its assembly of the lac-repressor headpiece at the solid level.

2.4.3 Varieties of Control Reasoning in BB1

As this trace illustrates, BB1 enables PROTEAN to perform a kind of hierarchical planning [16, 15, 46, 37] with two important differences. First, hierarchical planning systems typically refine selected plans to sequences of specific actions to be performed on specified sequences of problem-solving cycles. By contrast, a BB1 system can refine selected plans to any desired level of specificity. Thus, PROTEAN currently refines its plan to a sequence of two action classes, where each class is characterized by a set of desirable attribute-value relations. PROTEAN performs the "best" actions in each class during an open-ended problem-solving time interval that begins when a specified control state occurs and terminates when a specified solution state is achieved. Second, hierarchical planning systems typically formulate complete plans prior to beginning plan execution. By contrast, a BB1 system can--and generally does--construct its plan incrementally during plan execution, taking account of the results of previously executed actions in its reasoning about subsequent plan elements. Thus, PROTEAN does not generate its second focus until after it has achieved the goal of its first focus. PROTEAN uses the anchor established during its first focus to specify some of the heuristics under its second focus.

In addition to these extended capabilities for hierarchical planning, a BB1 system can perform other kinds of control reasoning.

First, since BB1 generates its control plan incrementally and explicitly represents the evolving control plan on the control blackboard, a system can interrupt, depart from, modify, discard, or resume construction and execution of a plan in response to the dynamic situation. For example, PROTEAN could begin implementing the Develop-PS-of-Best-Anchor strategy illustrated in the trace above, but subsequently determine that it had chosen a suboptimal anchor. A control knowledge source triggered by this observation could "back up" PROTEAN's control reasoning to its first focus (Create-PS-of-Best-Anchor), add a new heuristic to exclude the originally chosen anchor, and then allow PROTEAN to resume its problem-solving activities in accordance with the modified control plan.

Second, in addition to the top-down inference method underlying skeletal planning, a BB1 system can incorporate a variety of other inference methods, such as: (a) bottom-up methods that hypothesize the desirability of pending actions not explicitly favored by the current control plan; (b) goal-directed methods that plan actions whose results would trigger actions favored by the current control plan; and (c) opportunistic methods that plan actions whose results would improve a targetted aspect of the current solution. Taking goal-directed methods as an example, suppose PROTEAN's control plan favored actions involving the knowledge source Yoke-Structures at a time when no such actions appeared on its agenda. A control knowledge source triggered by this situation could determine that Yoke-Structures is triggered by modifications to an anchoree's *locations* and that the knowledge sources Anchor-Helix and Anchor-Coil produce such modifications. It would record a heuristic favoring actions involving these knowledge sources.

Finally, a BB1 system integrates reasoning about control of all domain and control actions within a uniform blackboard architecture. Thus, for example, PROTEAN records and concurrently applies heuristics favoring control actions over domain actions along with its strategic heuristics favoring particular kinds of domain actions.

All of these capabilities are discussed in more detail and illustrated in [23]. We currently are evaluating more complex PROTEAN strategies that exploit some of the capabilities for control reasoning.

2.5 Explanation in BB1

2.5.1 Overview of Explanation

BB1's explicit representation of a system's control plan provides a database for use in explaining a system's actions. A system can explain its actions by presenting information from the *description* and *rationale* attributes of selected elements of the control plan. Currently, a system can respond to two kinds of requests. It responds to the request "explain plan element *p*" by presenting the rationale stored with *p*. It responds to the request "why plan element *p*" by presenting the description stored with *p*'s superordinate plan element. The user can make either request with respect to any element of the control plan at any time during the interaction. However, a typical interaction proceeds bottom-up through the control plan, with requests for explanation at each level.

2.5.2 An Illustrative Explanation

Let us consider, for example, PROTEAN's explanation of its decision to schedule KSAR15 in the program trace discussed above (see Figure 22). The user asks PROTEAN to explain the scheduling decision. In response, PROTEAN presents the description stored with each of the heuristics underlying its scheduling decision. The user recognizes the value of most of the heuristics, but asks PROTEAN to justify heuristic2. PROTEAN presents heuristic2's rationale (see Figure 18). Next, the user asks PROTEAN why it is using the entire set of heuristics. In response, PROTEAN describes their superordinate focus (see Figure 17). The user asks PROTEAN to explain that focus and PROTEAN presents its rationale. Next, the user asks PROTEAN why it is using that focus. In response, PROTEAN describes the focus decision's superordinate strategy (see Figure 13). Finally, the user asks PROTEAN to explain the strategy and PROTEAN presents its rationale.

Insert Figure 22

2.5.3 Varieties of Explanation in BB1

As this example illustrates, BB1 exploits the structure of its control plan to provide a strategic explanation of its behavior. That is, it explains why it chooses to perform particular actions by showing how they fit into its larger plan for solving a problem. When a system has an elaborate, hierarchical strategy, BB1 uses that hierarchical structure to structure its explanations.

BB1 currently explains its behavior in terms of the description and rationale attributes of control decisions. However, there is a substantial amount of additional information available for explanation. For example, it could explain the goals being addressed by particular control decisions. It also could explain the triggering conditions that make particular KSARs feasible. We are currently working to add these kinds of information to BB1's explanations.

BB1 currently supports a demand-driven style of explanation. However, with additional knowledge, it could support other more intelligent styles of explanation as well. For example, suppose a domain expert had initiated a PROTEAN run that required application of a complex, but reliable strategy to a large protein. The expert might go off to other activities and return periodically to monitor PROTEAN's progress. In this situation, PROTEAN could provide a satisfactory explanation by simply reporting how far it had progressed in its strategy.

Alternatively, suppose a system programmer were experimenting with alternative control strategies to assess their computational requirements. The programmer might request explanation of many of PROTEAN's chosen actions. Given the frequency of questioning, PROTEAN might condense explanations that repeat parts of recent preceding explanations. For example, if it were asked to explain two consecutive actions that were supported by the same focus and heuristics, PROTEAN might simply observe that the action served the same focus, without repeating all of the heuristics. Since these different styles of explanation usurp the user's power to determine what information will be presented, we plan to implement them as sets of generic explanation knowledge sources that users may or may not choose to incorporate in particular application systems.

2.6 Learning in BB1

2.6.1 Overview of Learning

BB1 structures the data needed to learn new control strategies. Learning knowledge sources can observe relationships between KSARs, the events that trigger them, and the events that they produce. They can observe similarities and differences among competing KSARs and determine how those KSARs rate against the current control plan and against its constituent heuristics and Foci. They can exploit known data structures to program new control knowledge sources. For example, a generic learning knowledge source called MARCK [24] learns a new control heuristic whenever a domain expert corrects an application system's scheduling decisions. The following section illustrates MARCK's learning of a new control heuristic for PROTEAN.

2.6.2 An Illustrative Example of Learning

Figure 23 illustrates how MARCK goes about learning the heuristic "Prefer-Anchoring-over-Yoking." MARCK is triggered when a domain expert overrides PROTEAN's decision to execute KSAR56 and instructs it to execute KSAR55 instead. MARCK hypothesizes that the expert is using a control heuristic that distinguishes the two KSARs, but is not included under the current focus. It assumes further that the target heuristic focuses on some difference between the two KSARs and sets about identifying that difference. Comparing the two KSARs, MARCK rules out attributes on which: (a) the two KSARs have identical values; and (b) current control heuristics favor KSAR55 over KSAR56. If more than one attribute remains, MARCK must ask the domain expert which is the key attribute. It then draws an appropriate *canonical function* from its library, instantiates it for the identified attribute, translates the function into an English-language description, and requests an English-language rationale from the expert. Finally, MARCK records its accumulated information as a new heuristic on the control blackboard and programs a new control knowledge source to generate it on future PROTEAN runs.

Insert Figure 23

2.6.3 Varieties of Learning in BB1

BB1 provides a rich foundation of data for many different kinds of learning.

First, in addition to MARCK, we are exploring other learning procedures that can operate on the behavioral data explicated in a BB1 system. For example, we are working on a collection of learning knowledge sources called WATCH [20]. These knowledge sources observe a domain expert scheduling a system's problem-solving actions and recursively abstract a hierarchy of control heuristics that capture sequential regularities in the expert's scheduling decisions. Then they automatically program new control knowledge sources that post and expand the hierarchy top-down during subsequent problem-solving episodes.

Second, these and other learning procedures can be applied to BB1's explicit representations of events, actions, states, and facts, and to the various relationships among them. For example, a learning knowledge source could monitor the events produced by a particular knowledge source and, depending upon whether they typically contributed to accurate solutions, it could adjust the knowledge sources estimated *reliability*. Another knowledge could operate in a similar fashion to adjust a knowledge source's estimated *cost*.

2.7 Features of the BB1 Architecture

In summary, BB1 has several noteworthy features as an architecture for intelligent systems:

1. It provides a general blackboard mechanism for problem solving.

2. It supports three fundamental varieties of reasoning about action: control, explanation, and learning.

3. It provides a rich foundation and flexible capabilities for reasoning in each category.

4. Its basic three-step cycle is simple and transparent.

5. It can incorporate user-specified knowledge sources to expand and improve upon its current capabilities.

Thus, BB1 is a general architecture for intelligent systems that attack a variety of problem classes, with a variety of problem-solving methods, in a variety of subject-matter domains.

3. An Illustrative Framework: ACCORD

3.1 Overview of the Arrangement-Assembly Task

As discussed in section 1, a framework defines the actions, events, states, and facts involved in solving a particular class of problems by means of a particular method, but independent of subject-matter domain. For example, we have abstracted and extended the arrangement-assembly framework implicit in the original PROTEAN implementation and implemented it as the ACCORD knowledge base.

The protein-structure analysis problem that PROTEAN solves exemplifies a class of *arrangement problems*. These problems require the problem-solver to arrange a set of symbolic objects in some context to satisfy a set of constraints. PROTEAN works in the domain of protein chemistry. It must arrange protein structures, such as helices and random coils, in the three-dimensional physical space of the protein molecule. However, arrangement problems also arise in a variety of other domains, such as furniture arrangement, travel planning, and task scheduling.

In principle, a problem solver could use any of several different methods to solve a given arrangement problem. Table 3 lists a few of these methods. In practice, however, the problem solver may not have the knowledge necessary to apply particular methods to particular problems. For example, PROTEAN cannot apply the *selection*, *refinement*, *modification*, or *generation* methods because it does not have knowledge of alternative protein structures, a prototypical protein structure, almost-correct protein structures, or an algorithm for generating complete protein structures. Therefore, PROTEAN *constructs* hypothetical protein structures by means of the *assembly method* discussed in this paper. Unlike the other methods in Table 3, the assembly method <u>can</u> be applied to any arrangement problem. As we show later in the paper, ACCORD is an appropriate framework for systems that assemble arrangements in a variety of subject-matter domains.

Insert Table 3

The following sections describe the knowledge content and representational conventions of BB1 frameworks in general and of ACCORD in particular: (a) the conceptual network that organizes all framework level knowledge; (b) types of domain entities; (c) role types for solution elements; (d) types of actions, events, and states; (e) characteristic relations among actions, events, and states; (f) linguistic templates for actions, events, and states; (g) the partial matches among templates; and (h) translations of framework templates into BB1 templates. The final section below describes the BB1 *framework-interpreter*--a collection of generic procedures for operating on framework knowledge structures.

3.2 The Conceptual Network

We represent all of the knowledge in a framework within a conceptual network [48]. As illustrated in Figure 24, the network distinguishes three kinds of concepts: types, individuals, and instances.

Concept *types* intensionally define the generic concepts that figure in a task by means of is-a links. *Role types* define the roles played by entities in solutions within a particular task. For ACCORD, these are the *arrangement-roles* played by the objects involved in arrangement-assembly tasks. *Natural types* define the *actions, events,* and *states* that figure in a task. For ACCORD, these are *assembly* actions, events, and states. Natural types may also specify task-specific concept types with which to define task-relevant domain entities. For ACCORD, these are the concept types: *object, constraint,* and *context*. Each of these concept types is discussed in detail below.

Concept *individuals exemplify* particular concept types. Concept *instances instantiate* particular individuals and *play* particular roles in particular contexts. The network may specify additional relations among concepts. For example, one concept may *include* a number of constituent objects. These and all other links in the network have corresponding inverse links: *can-be-a, is-exemplified-by, is-instantiated-by, is-played-by,* and *is-included-by.* (In Figure 24, and elsewhere in the paper, we use [bracketed] link names to indicate legal links between kinds of entities and unbracketed link names to indicate actual links between actual entities).

Insert Figure 24

The distinction among concept types, individuals, and instances corresponds roughly to the distinction among the generic concepts of a domain or task (e.g., a helix), the specific objects involved in a particular problem (e.g., helix1, the first helix in the primary sequence of the lac-repressor headpiece), and instances of those objects involved in particular hypothetical solutions to the problem (e.g., helix1-1, that is, helix1 in its role as anchor of partial arrangement pa1). For example, Figure 24 expresses these and other PROTEAN facts:

```
Helix1-1 is-a instance.
Helix1-1 plays anchor.
Anchor is-a role type.
Helix1-1 instantiates helix1.
Helix1 is-a individual.
Helix1 includes amino-acid35.
Helix1 exemplifies helix.
Helix is-a secondary-structure.
Secondary-structure is-a object.
Object is-a natural type.
```

An implicit *$is-a* relation holds between any two concepts related by a chain of instantiates, exemplifies, and is-a links. For example, we may infer that:

```
Helix1-1 $is-a secondary-structure.
```

because:

```
Helix1-1 instantiates helix1.
Helix1 exemplifies helix.
Helix is-a Secondary-Structure.
```

Similarly, a *$includes* relation holds between concepts related by a chain of instantiates, exemplifies, is-a, and includes links. For example, we may infer that:

Helix1-1 *$includes* Amino-Acid35.

because:

```
Helix1-1 instantiates helix1.
Helix1 includes Amino-Acid35.
```

A *$plays* relation holds between concepts related by a chain of exemplified-by, instantiated-by, and plays links. For example, we may infer that:

```
Helix $plays anchor.
```

because:

```
Helix is-exemplified-by helix1.
Helix1 is-instantiated-by helix1-1.
Helix1-1 plays anchor.
```

These and all other *$<link>* relations have corresponding inverse relations that hold between corresponding chains of inverse component relations. For example, we may infer that:

```
Anchor $is-played-by helix.
```

because:

```
Anchor is-played-by helix1-1
helix1-1 instantiates helix1.
helix1 exemplifies helix.
```

Any concept in the network may specify particular attributes, along with static or procedural values. For example, PROTEAN's concept network includes the facts that: helix has an attribute called *shape*, whose value is cylinder; and secondary-structure has an attribute called *length*, whose value is determined by a procedure called Number-of-AA that counts the number of amino-acids included by the secondary-structure. Like relations among concepts, these attributes are inheritable. For example, helix1-1's shape is cylinder and its length is determined by the procedure Number-of-AA.

One class of attributes warrants special mention. *Modifiers* are attributes whose procedural attachments evaluate the applicability of the named descriptors to any given concept individuals or instances. For example, PROTEAN's modifier *long* is an attribute of the concept type secondary-structure. Its value, which is computed by the procedure called How-Long-Is, is a function of the number of amino-acids included by a particular secondary-structure (that is, by a particular alpha-helix, beta-sheet, or random-coil). All such procedures return numerical values scaled 0-100, where 0 signifies minimal applicability of the modifier and 100 signifies maximal applicability. However, a framework can distinguish two different procedural definitions for each modifier.

Threshold procedures evaluate concepts in an all-or-none fashion. For example, PROTEAN might refer to a "long helix," meaning "a helix that has at least 15 amino acids." An individual helix, say helix1, either matches this description or it does not. Therefore, the threshold procedure attached to the attribute long returns a value of 100 for any helix that includes more than 15 amino acids and a value of 0 for any helix that includes fewer than 15 amino acids. In general, threshold procedures return a value of 100 or 0, depending upon whether or not the modified concept exceeds a designated threshold on a designated attribute.

Scale procedures evaluate concepts in a graded fashion. For example, PROTEAN might refer to a "long helix," meaning "a helix that includes at least 15 amino acids is better than one that includes 10-14 amino acids, which is better than one that includes fewer than 10 amino acids." An individual helix, say helix1, matches this description to some degree. Therefore, the scale procedure attached to the attribute long returns a value of 100 for any helix that includes more than 15 amino acids, a value of 50 for any that includes 10-14 amino acids, and a value of 0 for any helix that includes fewer than 10 amino acids. In general, scale modifiers return values somewhere in the range 0-100, depending upon the degree to which the modified concept exhibits a designated attribute.

Threshold or scale procedures may be specified within an expression by extending the modifier name with "-T" or "-S." However, as discussed below, BB1 knows in which circumstances each type of procedure typically applies. If no extension appears in a modifer, it uses the appropriate procedure.

3.3 Types of Domain Entities

A framework provides skeletal branches of the natural-type hierarchy in which to define relevant domain entities.

For the arrangement-assembly task, ACCORD provides skeletal branches for: the *objects* to be arranged, the *context* in which the objects must be arranged, and the *constraints* that must be satisfied within the arrangement. Particular constraints may *involve* particular objects and constraints. Figure 25 illustrates how PROTEAN instantiates these skeletal branches with biochemistry entities. In addition, PROTEAN specifies the characteristic attributes of and relations among entities. For example, it specifies that alpha-helix, beta-sheet and random coil have the attribute *shape*, with the values cylinder, prism, and sphere, respectively.

Insert Figure 25.

3.4 Role Types

A framework defines the roles that problem entities can play in hypothetical solutions.

ACCORD defines the arrangement roles illustrated in Figure 26. An *arrangement* is a potential complete solution to an arrangement problem, that comprises one or more partial-arrangements that, together, comprise a criterial subset of its objects, constraints, and context. A *partial-arrangement* is a partial solution to a problem, that comprises a non-criterial subset of its objects, constraints, and context. An *included-object* is one of the objects from the problem that has been selected for inclusion in a partial-arrangement. Included-object has three subordinate subtypes. An *anchor* is an included-objects that has been assigned a fixed location to define the local context of a partial arrangement. An *anchoree* is an included-objects that has at least one constraint with the anchor. An *appendage* is an included-objects that has at least one constraint with at least one anchoree.[4]

Figure 26 also illustrates characteristic relations among solution elements that play particular roles. An arrangement *includes* partial-arrangements, which, in turn, *include* Included-objects. Anchors *anchor* anchorees. Anchorees may *append* appendages. Two included-objects may *yoke*

---

[4]We have not yet found it necessary to elaborate similar role types for constraints and contexts, but we may do so in the future.

one another. Three or more included-objects may *consolidate* with one another. A partial-arrangement may incorporate, merge, or dock another one.

Finally, ACCORD specifies a number of characteristic attributes and default values for solution elements that play particular roles (not shown in Figure 26). For example, included-object has a *locations* attribute, whose default value is Nil, that specifies its legal locations in its partial-arrangement context, given the constraints that have been applied at any point in time. Included-object also has an attribute named *secure* whose value is a procedure for rating (0-100) the degree to which an included-object's current locations have been restricted.

3.5 Types of Actions, Events, and States

A framework defines task-specific action, event, and state types as homologous variations on an underlying network of root verbs.

Insert Figure 27

ACCORD defines the type hierarchy of root verbs shown in Figure 27. The top-level verb, *assemble,* means: solve an arrangement problem by means of the assembly method. Assemble has four subtypes. *Define* means: construct a partial arrangement that includes particular objects in particular roles. *Position* means: identify the locations in which particular objects can lie within a particular partial arrangement while satisfying particular constraints. *Coordinate* means: identify the locations in which particular objects can lie within a partial arrangement while satisfying their part-whole relations with previously positioned superordinate or subordinate objects. *Integrate* means: combine two partial arrangements to form a single, larger partial arangement. Each of the four verb subtypes--define, position, coordinate, and integrate--has two or more subordinate subtypes, as described below.

Define has three sub-types. *Create* means: record a blackboard objects representing a new partial arrangement. *Include* means: create instances of particular objects or constraints within a particular partial arrangement. *Orient* means: declare that a particular objects in a partial arrangement is the anchor and assign the roles anchoree and appendage to other included objects depending upon whether or not they have constraints with the anchor.

Position has five subtypes. *Anchor* means: identify the locations in which an anchoree satisfies particular constraints with the anchor. *Append* means: identify the locations in which an appendage satisfies constraints with an anchoree or appendage that has already been positioned. *Yoke* means: prune the locations for two included-objects that have already been positioned so that they include only locations in which the two objects satisfy constraints with one another. *Restrict* means: prune the locations identified for an anchoree or appendage to include only those that satisfy additional constraints. *Consolidate* means: prune the locations for three or more objects to include only those that satisfy all constraints among the objects simultaneously.

Coordinate has two subtypes. *Refine* means: identify locations for a previously positioned objects's constituent objects so as to satisfy their part-whole relationship. *Adjust* means: identify an objects's locations to satisfy its part-whole relationship with previously positioned constituent objects.

Integrate has three subtypes. *Merge* means: combine two partial arrangements that have the same anchor. *Incorporate* means: combine two partial arrangements that include anchorees or appendages. *Dock* means: combine two partial arrangements that have no common objects, but include objects that constrain one another.

ACCORD also specifies entailments of these root verbs (see Figure 28). For example, the anchor verb *entails* the *generate* verb, which means: generate a family for an included-object. Similarly, the position verb entails the *apply* verb, which means: apply a constraint to an included-object within a partial arrangement. An implicit *$entails* relation holds between two concepts related by any chain of is-a, exemplifies, instantiates, and entails links. For example, we may infer that:

```
Anchor $entails apply.
```

because:

```
Anchor is-a Position.
Position entails apply.
```

Conversely, we may infer that:

```
Apply $is-entailed-by anchor.
```

because:

```
Apply is-entailed-by position.
Position can-be-a anchor.
```

A framework distinguishes homologous type hierarchies for actions, events, and states by different verb tenses: *Do-verb* signifies an action. *Did-verb* signifies an event. *Is-verbed* signifies a state. As illustrated in Figure 28, all relations and attributes in the root verb hierarchy reappear in the action, event, and state type hierarchies. A framework also recognizes implicit states reflecting the existing properties of particular concepts (e.g., Has helix2 shape cylinder) and the relationships between them (e.g., Exemplifies helix1 helix). As a consequence, the number of recognizable state types in an application greatly exceeds the number of action and event types. For reasons of efficiency, a framework does not explicitly enumerate all such states, but only those that have important relationships (e.g., is-caused-by, is-entailed-by) to actions, events, or states in the type hierarchy. Nonetheless, it supports verification and assessment of all explicit and implicit states in the conceptual network.

Insert Figure 28)

3.6 Relations among Actions, Events, and States

A framework specifies legal relations among different types of actions, events, and states [1, 36]. Events of a particular type can *trigger* actions of a particular type, that is, indicate that the actions are potentially feasible. States of a particular type can *enable* triggered actions of a particular type, that is, render the triggered actions feasible. Actions of a particular type can *cause* events of a particular type. Finally, events of a particular type can *promote* states of a particular type. Figure 29 illustrates some of the legal relations specified in the ACCORD knowledge base.

Insert Figure 29

An implicit *$<links>* form of each of these relations:

```
A [$<links>] B
```

holds for any two concepts, a and b, whenever:

```
a $is-a A or a $entails A.
          and
B $is-a b or B $entails b.
```

For example, we may infer from Figure 29 that:

```
Did-anchor [$triggers] do-yoke.
```

because:

```
Did-anchor is-a did-position.
Did-position [triggers] do-yoke.
```

Similarly, we may infer that:

```
Do-yoke [$causes] did-apply.
```

because:

```
Do-yoke [causes] did-yoke.
Did-yoke $entails did-apply.
```

Note that legal relations such as those specified in Figure 29 may not actually hold among all individual actions, events, and states of the specified types. For example, a did-position event can trigger a do-yoke action. But an individual did-position event may require additional attributes (discussed below) in order to trigger an individual do-yoke action.

3.7 Linguistic Templates for Actions, Events, and States

A framework provides linguistic templates for all root verbs and their entailments. Each template comprises a *verb keyword*, followed by a specified sequence of *formal parameters*, interspersed with optional conjunctions and prepositions (noise words). Particular actions, events, or states are represented as *patterns* that instantiate the formal parameters of particular templates with particular concept types, individuals, or instances. In addition, each keyword and formal parameter value in a pattern may be preceded by any number of modifiers and followed by a local variable name in parentheses.

Table 4 shows ACCORD's templates for the arrangement-assembly root verbs. (For brevity, we omit ACCORD's templates for entailed verbs.) For example, the anchor template is:

```
Anchor anchoree to anchor in pa with constraint.
```

Here the keyword, anchor, is followed by the sequence of formal parameters: anchoree, anchor, pa, constraint, with some parameters preceded by the declared noise words: to, in, with.

A system instantiates these templates with domain-specific entities to form particular action, event, and state patterns. For example, PROTEAN might instantiate the anchor template as this action pattern:

```
Do-anchor helix2-1 to helix1-1 in pa1 with NOE1.
```

PROTEAN could represent a larger class of actions with this pattern:

```
Do-anchor helix to helix1-1 in pa1 with constraints.
```

It could represent a restricted class of actions by inserting modifiers before some parameter values, as in this pattern:

```
Quickly do-anchor long helix to helix1-1 in pa1 with strong constraints.
```

PROTEAN could instantiate event and state patterns in a similar fashion by substituting the appropriate did-verb or is-verbed forms of the root verbs.

Insert Table 4

3.8 Partial Matches Among Templates

A framework defines the potential partial matches among action, event, and state patterns by identifying corresponding parameters in their underlying templates. (These correspondences need not be one-to-one.) Two patterns match to the degree that the values of their corresponding parameters match. For example, Figure 30 identifies corresponding parameters in the assemble, position, and anchor template.

Insert Figure 30

Consider the position and anchor templates. By definition, the two keywords, position and anchor, correspond. In this context, the formal parameters included-object and anchoree correspond because they both represent objects that the actions position. The two formal parameters called pa correspond because they both represent the partial arrangement in which the actions occur. The two formal parameters called constraints correspond because they both represent constraints that the actions apply. The anchor template's formal parameter called anchor does not correspond to anything in the position template because the position template does not specify an object that lies at the center of the designated local coordinate system.

Given this knowledge, a system can assess the degree to which two patterns match by assessing the matches between their formal parameter values. For example, PROTEAN can assess the degree to which the pattern:

```
Anchor helix2-1 to helix1-1 in pa1 with NOE1.
```

matches the pattern:

```
Position long helix in pa1 with strong constraint.
```

by assessing the matches of:

```
anchor against position;
helix2-1 against long helix;
pa1 against pa1;
NOE1 against strong constraint.
```

3.9 Framework-BB1 Template Translations

Since a framework exists in the context of the BB1 architecture, it provides the knowledge necessary to translate certain framework templates into semantically equivalent BB1 templates and vice versa. So far, we have found it necessary to provide such knowledge for terminal action patterns and for all state patterns. In both cases, translation knowledge comprises the parameterized framework templates and the corresponding parameterized BB1 templates, with corresponding parameters of the same names. Thus, BB1 can translate patterns between representations by means of a variable-substitution procedure discussed below.

For example, Figure 31 shows the BB1 template for the do-anchor action. As this example illustrates, each BB1 action template is a parameterized program of rules that evaluate lisp expressions, set local variables, and modify objects on the blackboard or in the knowledge base. (Note that all application-specific routines for constraint satisfaction are inserted indirectly through calls to ACCORD's generic *CSS-<extension>* functions.) Both do-anchor templates refer to the parameters: anchoree, anchor, pa, and constraints.

Insert Figure 31

Figure 32 shows the BB1 template for the is-anchored state. As this example illustrates, each BB1 state template is a parameterized program of blackboard access functions. Both is-anchored templates refer to the parameters: anchoree, anchor, pa, constraints.

Insert Figure 32

In addition to these explicitly stored state translations, BB1 automatically translates any has-attribute state pattern instantiating the prototypical framework template:

```
Has object attribute value
```

into the equivalent prototypical BB1 template:

```
(Equal ($Value object attribute) value).
```

3.10 The BB1 Framework-Interpreter

We have extended BB1 with a *framework-interpreter*: a collection of procedures for *parsing* patterns, *matching* patterns, *quantifying* the match between two patterns, *generating* an ordered list of quantified instantiations of a pattern, and *translating* framework patterns into BB1 patterns. In all cases where these procedures apply, BB1 can use either the new procedures or its standard procedures (applicable to BB1 knowledge structures), as appropriate. The BB1 framework-interpreter applies to <u>any</u> <u>user-specified</u> <u>framework</u> defined with the BB1 knowledge structures illustrated above for ACCORD.

3.10.1 Parsing Patterns

The BB1 *parser* converts patterns from their English form to a parsed form for use by the matcher, quantifier, generator, and translator. The parser first removes noise words (conjunctions and prepositions) from a pattern. It then works left to right, using recognized verb keywords and the sequence of parameters in their associated templates to identify the pattern's constituent phrases. The parser produces a list of simple lists, each of which contains a single parameter value and the modifiers that precede it in the pattern. For example, the parser would parse the pattern:

```
Quickly do-anchor long helix to helix1-1 in pa1
        with strong constraint.
```

as the list:

```
((do-anchor Quickly)
 (helix long)
 (helix1-1)
 (pa1)
 (constraint strong))
```

Other interpretation procedures access particular parameter phrases according to their sequential positions in the templates and parsed lists.

3.10.2 Matching Patterns

The BB1 *matcher* assesses whether a *test pattern* matches a *target pattern*. For each corresponding parameter in the two patterns, the matcher declares a match whenever the test pattern value has a $is-a, $entails, or $plays relation with the target pattern value. A *perfect match* is one in which the matcher declares a match for all parameters (verbs and nouns) in the target pattern. However, the matcher uses the partial-match knowledge described above to assess the partial match between <u>any two patterns</u>, regardless of the number of corresponding parameters between them. Figure 33a illustrates a perfect match between two PROTEAN action patterns.

Insert Figure 33

3.10.3 Quantifing a Match

The BB1 *quantifier* records a numerical assessment of the match between each parameter value in a test pattern and: (a) its corresponding parameter value in a target pattern; and (b) each modifier of the corresponding parameter value in the target pattern. It records 0 for each non-matching parameter value and 100 for each matching parameter value. For non-matching parameters, the quantifier also records 0 for each modifier of the parameter value in the target pattern. For matching parameter values, it records for each modifier a number between 0 and 100, which it obtains from the attribute named by the modifier. A *perfect quantified match* is one in which the test pattern receives a value of 100 for all parameters in the target pattern and their associated modifiers. Again, however, the quantifier numerically assesses the degree of match between any two patterns regardless of the number of corresponding parameters. Figure 33b illustrates a quantified match between two PROTEAN action patterns.

As discussed above, modifiers may specify threshold or scale procedures with the extensions "-T" or "-S" to the modifier name. However, BB1 knows in which circumstances threshold and scale procedures typically apply and uses the appropriate one if no extension appears in the named modifier. For example, BB1 uses threshold procedures to quantify matches underlying its all-or-none triggering decisions and scale procedures to quantify matches underlying its graded ratings of pending KSARs.

3.10.4 Generating an Ordered List of Quantified Matches

The BB1 *generator* generates all (or a specified number of) values for a designated parameter that legally instantiate a set of patterns or *phrases*. The generator first follows links in the concept network to find values that match parameter values and associated threshold modifiers and relations specified in the input patterns. It then rates each value against associated scale modifiers in the input patterns. It returns all values and their ratings, "best first." For example, Figure 34 illustrates generation of all long helices that are positioned in some partial arrangement.

Insert Figure 34

3.10.5 Translating Between Framework and BB1 patterns

The BB1 *translator* uses a variable-substitution procedure to translate framework and BB1 patterns into one another. For example, Figure 35 illustrates the translation of an ACCORD pattern for the do-anchor action into the semantically equivalent BB1 action pattern.

Insert Figure 35

----

4. Reasoning within the BB* Environment

In this section, we show how a framework such as ACCORD enhances the representation and performance of a BB1 application system such as PROTEAN.

4.1 Domain Reasoning in BB*

4.1.1 Domain Knowledge Sources: Representation and Computation

Given a framework, a domain knowledge source can represent each of its main components--trigger, context, precondition, obviation condition, and action--as event, state, and action patterns that exemplify particular event, state, or action types. For example, Figure 36 shows the ACCORD representation of PROTEAN's knowledge source Yoke-Structures (see Figure 10). Given this representation, BB1 can exploit its framework-interpretation procedures during knowledge source invocation and execution. The following sections discuss the representation and processing of these knowledge source components: trigger, context, precondition, action, and result.

Insert Figure 36

4.1.2 Knowledge Source Trigger

A knowledge-source trigger comprises one or more event patterns. BB1 triggers a knowledge source for a given blackboard event if it assesses a perfect quantified match of the blackboard event against each of the knowledge source trigger's event patterns. At the same time, it binds the value of each parameter in the trigger patterns to the specified local variable name (or, if none is specified, to an internally generated name).

For example, Yoke-Structures's trigger comprises one did-restrict event pattern:

```
Did-restrict included-object (yokee) in any-pa (the-pa).
```

Suppose the following blackboard event occurred:

```
Did-anchor helix2-1 to helix1-1 in pa1 with NOE1.
```

This event produces a perfect quantified match to Yoke-Structures's trigger pattern because:

```
Did-anchor $entails did-restrict.
Helix2-1 $plays included-object.
Pa1 $is-a pa.
```

Therefore, BB1 would trigger Yoke-Structures and bind the two local variables: yokee to helix2-1 and the-pa to pa1.

4.1.3 Knowledge Source context

A knowledge-source context comprises a nested set of expressions of the form:

```
For <variable> in <state patterns>.
```

For each such expression, BB1 generates and identifies as a context each unique combination of variable-value pairs that match the pattern. If several such expressions are nested, BB1 applies this procedure recursively. It generates a KSAR for each identified context and places all generated KSARs on the agenda.

For example, Yoke-Structures's context comprises two expressions:

```
For partner in:
  Includes the-pa partner.
  Not Is yokee partner.
For constraint in:
  Involves constraint yokee.
  Involves constraint partner.
```

Let us continue the example begun above. Based on the first expression, BB1 generates alternative values of the context variable, partner: all objects that are included by pa1 (the-pa), excluding helix2-1 (yokee). Supposing that pa1 includes one such object, BB1 generates one value of partner: helix3-1. Based on the second expression, BB1 generates for each value of partner alternative values of the context variable, constraint: all constraints that involve helix3-1 (partner) and helix2-1 (yokee). Supposing that two such constraints exist, BB1 generates two values for constraint: NOE6 and NOE8. Finally, BB1 generates a unique context representing each combination of context-variable values and generates a separate KSAR for each context, for example KSAR 50 in Figure 37. (Figure 11 shows the same KSAR represented in BB1 knowledge structures).

Insert Figure 37

4.1.4 Knowledge Source precondition

A knowledge-source precondition comprises any number of state patterns that must match information on the blackboard or in the knowledge base before the KSAR can execute its action. For each KSAR, BB1 translates and evaluates each precondition pattern, performing specified variable bindings along the way. If all preconditions evaluate to true, BB1 places the KSAR on the agenda of executable actions. If any do not evaluate to true, BB1 places the KSAR on the agenda of triggered actions and rechecks unsatisfied preconditions on each cycle until all are true.

For example, Yoke-Structures's precondition:

```
Has partner locations.
```

specifies that Yoke-Structures can execute its action only when the previously identified partner has an attribute named locations whose value is not nil. For KSAR50 above, BB1 translates this pattern into the BB1 pattern:

```
($Value helix3-1 locations)
```

and evaluates it. If it evaluates to true, BB1 determines that KSAR1 is executable.

4.1.5 Knowledge Source Action

A knowledge-source action is a terminal action pattern whose parameters are bound within a KSAR during the triggering, context-matching, and precondition-evaluation procedures described above. When BB1 decides to execute a particular KSAR, it translates the action pattern into the equivalent BB1 action and sends it to BB1's low-level action interpreter.

For example, KSAR50 specifies the action pattern:

```
Do-yoke helix2-1 with helix3-1 in pa1 with cset1.
```

BB1 translates this pattern into the equivalent BB1 action pattern (illustrated in Figure 35) and sends it to the low-level action interpreter for execution.

4.1.6 Knowledge Source Result

A knowledge source result is a terminal event pattern that corresponds exactly to the knowledge source action pattern. Within a KSAR, corresponding parameters in the action pattern and result pattern have identical values. When BB1 executes the action of the KSAR, it generates the event pattern and records it on its internal *event list* for use during knowledge source triggering.

For example, in executing KSAR50, BB1 generates the event pattern:

```
Did-yoke helix2-1 with helix3-1 in pa1 with cset1.
```

(Figure 12 shows the same event represented in BB1 knowledge structures.)

4.1.7 Advantages for Domain Reasoning

As these examples illustrate, a framework such as ACCORD improves domain knowledge sources and domain reasoning in several ways:

1. There is a clean distinction between domain-specific conceptual knowledge and task-specific problem-solving actions.

2. Domain knowledge sources and KSARs are shorter, simpler, more perspicuous, and more machine-interpretable than they are when expressed in the standard BB1 knowledge structures.

3. It is easier to program domain knowledge sources in the English-like language of a framework than to program them in Lisp expressions.

4. The well-structured declarative representation of action, event, and state types provides a natural discrimination network for efficient triggering, context binding, and precondition matching procedures.

4.2 Control Reasoning in BB*

We discuss the following aspects of control reasoning: specification of control knowledge sources, representation of focus decisions, rating KSARs against focus decisions, representation of of abstract control plans, and reasoning about control plans. We summarize the advantages of using a framework for all aspects of control reasoning at the end of the section.

4.2.1 Control Knowledge Sources: Representation and Computation

A framework permits control knowledge sources to represent their main components in terms of event, state, and action patterns. For example, Figure 38 shows the control knowledge source: Append-to-Secure-Anchorees.

Insert Figure 38

Control knowledge sources undergo the same invocation procedures described above for domain knowledge sources. For example, the event:

```
Did-anchor helix2-1 to helix1-1 in pa1 with NOE1.
```

in which helix2-1 was restricted to a criterially small number of locations would produce KSAR51, shown in Figure 39. When executed, the KSAR would record a new focus with the specified attributes.

Insert Figure 39

4.2.2 Representing Focus Decisions

As discussed in section 2 above, a focus decision identifies a class of actions that are desirable during designated time intervals. It has two key computational attributes: A *prescription* defines a desirable class of actions. A *goal* defines a termination condition for the focus. Given a framework such as ACCORD, a system can represent the prescriptions and goals of its focus decisions as instantiated action, event, and state patterns.

For example, if the Append-to-Secure-Anchorees KSAR in Figure 39 above, were executed, it would produce the focus decision shown in Figure 40. Notice that the prescription of this focus:

```
Perform:
  Do-append appendage to helix2-1 in pa1 with constraint.
```

captures the meaning of several heuristics as represented in standard BB1 knowledge structures:

```
Prefer KSARs that execute append actions.
Prefer KSARs that operate on appendages.
Prefer KSARs that position something relative to helix2-1.
Prefer KSARs that operate in the context of pa1.
Prefer KSARs that apply constraints.
```

Similarly, although the goal in this example represents a single blackboard access function, the goals of other focus decisions could capture the meaning of a program of such functions.

Insert Figure 40

More generally, a focus prescription can specify desirable actions in terms of the actions themselves, the events that trigger them, the states that enable them, the events they cause, or the states they promote. Table 5 shows examples of these other kinds of prescriptions. Similarly, a focus goal can specify desirable conditions in terms of any state of the knowledge base or any blackboard. Table 6 shows examples of other kinds of goals. This flexibility supports the integration of different inference methods (e.g., data-driven, goal-driven). More importantly, it articulates the semantics of these methods in the context of control planning.

Insert Table 5

Insert Table 6

4.2.3 Rating Feasible Actions

Given a framework, a system can rate alternative feasible actions (pending KSARs) against desirable actions (current focus decisions) by means of the quantified match procedure discussed in section 3. It rates each parameter value in the KSAR against each corresponding parameter value and modifier in a focus decision. It combines these component ratings according to some integration function (either one specified in that particular focus or a default function) to produce a rating against the entire focus decision. Figure 33 above shows an example in which the KSAR action:

```
Do-anchor helix3-1 to helix1-1 in pa1 with NOE27.
```

is rated against the focus decision:

```
Perform:
  Do-position helix3-1 in pa1 with strong constraint.
```

4.2.4 Representing Abstract Control Plans

As discussed in section 2, abstract control plans result from a system's reasoning about general strategies encompassing variable problem-solving time intervals. Strategy decisions at one level of abstraction prescribe sequences of subordinate strategy decisions at lower levels. All control plans terminate in a sequence of focus decisions that the BB1 scheduler uses to rate pending KSARs.

Given a framework such as ACCORD, a system can represent a strategy's description and goal attributes as action, event, and state patterns, similar to those discussed above for focus decisions. Thus, strategy descriptions can specify desirable actions in terms of the actions themselves, the events that trigger them, the states that enable them, the events they cause, or the states they produce. Similarly, strategy goals can specify termination conditions in terms of any state of the knowledge base or blackboards. For example, Figure 41 shows the PROTEAN control plan from Figure 13 as represented in ACCORD knowledge structures. Figure 42 shows an excerpt from one of the more complex control plans we currently are evaluating. (Both of these figures show only the description attributes of component decisions.)

As these figures illustrate, the ACCORD representation clearly shows how each decision summarizes and prescribes its subordinates. In Figure 42, for example, the second sub-strategy decision:

```
Perform:
  Quickly do-position long constraining secondary-structure
    (target-object) in pa2 with strong constraints.
```

summarizes and prescribes its subordinate sequence of decisions:

```
Perform:
  Quickly do-position helix4-2 in pa2
    with strong constraints.

Perform:
  Quickly do-position helix6-2 in pa2
    with strong constraints.
```

because helix4-2 is the longest, most constraining secondary structure in pa2 and helix6-2 is the runner-up. Similarly, although it does not appear in Figure 42, the goal of the sub-strategy decision:

```
Has target-object few locations.
```

summarizes and prescribes the goal of its subordinate sequence of decisions:

```
Has helix4-2 few locations.

Has helix6-2 few locations.
```

Insert Figure 41

Insert Figure 42

4.2.5 Constructing the Control Plan

As discussed in section 2, a BB1 system dynamically contructs its control plan through the actions of knowledge sources that incrementally record and modify component control decisions. To support control reasoning, BB1 provides a growing repertoire of generic control knowledge sources that cooperate with application-specific control knowledge sources to generate control plans. (See, for example, the control knowledge sources Initialize-Prescription, Update-Prescription, and Terminate-Decision, discussed in section 2.)

A framework provides a richer foundation for generic control knowledge sources. For example, the control knowledge source called Refine-Parameters refines a strategy decision as a sequence of subordinate decisions by replacing one of its parameter phrases with legal values, best first. The strategy decision must specify which parameters to refine as the value of its attribute procedure-data. If the strategy specifies more than one parameter, Refine-Parameters applies recursively to each subordinate decision.

For example, although it does not appear in Figure 42, the strategy decision carries these additional attributes:

```
Description:
  Quickly do-position long constraining secondary-structure
  in current-best pa with strong constraints.
Procedure-type: Refine-Parameters
Procedure-data: (pa included-object)
```

The attribute procedure-type specifies that Refine-Parameters should refine this strategy. The attribute procedure-data specifies two parameters to be refined, pa and included-object. Beginning with the parameter pa, Refine-Parameters generates a subordinate by replacing the strategy's phrase, current-best pa, with the best legal value, pa1. It also removes the parameter pa from the subordinate's procedure-data. Moving on to the parameter included-object, Refine-Parameters generates a subordinate by replacing the phrase, long constraining secondary-structure, with the best legal value, helix3-1. Again, it removes the parameter included-object from the subordinate's procedure-data. Since this subordinate's procedure-data has a value of nil, Refine-parameters declares it to be a focus decision. The scheduler then uses the new focus to rate pending KSARs until the focus goal (refined from the strategy goal in the same manner) is satisfied.

When the focus goal is satisfied, Refine-Parameters is again activated. It returns to the focus decision's superordinate strategy and generates the next focus by replacing the phrase, long constraining secondary-structure, with the second best legal value, helix4-1. The scheduler uses the new focus to rate pending KSARs until its goal is satisfied.

When the new focus goal is satsified, Refine-Parameters is again activated. When it determines that there are no other legal values of the phrase, long constraining secondary-structure, it returns to the top-level strategy and generates the second subordinate strategy by replacing its phrase, current-best pa, with the second best value, pa2. It generates the first and second focus decisions under this sub-strategy as discussed above for the preceding sub-strategy.

Following BB1's emphasis on flexibility, Refine-Parameters can refine a strategy to an arbitrary level of detail. If a strategy specifies all of its parameters for refinement, each focus decision specifies the currently most desirable <u>individual</u> action. However, if a strategy specifies a subset of its parameters, as illustrated in Figure 43, each focus decision specifies the currently most desirable <u>class</u> of actions. In either case, the scheduler rates pending KSARs against the focus until its goal is satisfied.

As discussed above, BB1 can integrate multiple inference methods (e.g., top-down, bottom-up, goal-directed, opportunistic) in its reasoning about control plans. Refine-Parameters illustrates the advantages of using a framework for top-down strategy refinement. Similar advantages emerge in applying the other kinds of inference methods as well.

4.2.6 Advantages of Using a Framework for Control Reasoning

A framework provides the following advantages for control reasoning:

- It reinforces BB1's distinction between problem-solving actions and problem-solving strategies.

- It provides a simple, concise, perspicuous, and interpretable representation of control knowledge sources, KSARs, control decisions, and control plans.

- It provides a uniform representation for control decisions at all levels of abstraction.

- It enables individual control decisions to combine multiple heuristics while preserving their modularity.

- It explicitly and unambiguously articulates task-specific control parameters and the relationships among them, thereby enforcing a semantically correct mapping between KSAR attributes and the control heuristics used to rate them.

- It supports control knowledge sources that implement a variety of inference procedures (e.g., goal-driven, event-driven) and articulates the semantics of these different methods in the context of control planning.

- It provides a rich foundation for powerful generic control knowledge sources, thereby reducing the number of specific knowledge sources required for an application.

4.3 Explanation in BB*

As discussed in section 2, BB1 currently constructs explanations out of the description attributes of relevant control decisions and the links among them. Given a framework, these descriptions represent particular action, event, and state patterns. For example, Figure 43 shows how PROTEAN would explain its decision to perform KSAR55 based on the control plan excerpted in Figure 42. As illustrated in this example, a framework enhances the quality of BB1's explantions in four ways.

First, a framework provides an orderly and perspicuous language of explanation. The parameters of framework templates represent key task-specific control parameters that are instantiated with domain-specific concepts. The hierarchical organization of framework templates and their domain-specific values correspond to the hierarchical aspects of the problem-solving process.

Second, a framework provides a structured account of the organization of individual heuristics within a control decision and the KSAR attributes to which they are applied. For example, in Figure 43, PROTEAN clearly communicates that "helix6-2" was evaluated against several heuristics represented by the words: long, constraining, and secondary-structure.

Third, a framework provides these same advantages for explanations of the feasibility of recommended actions and for the goals of particular control decisions. For example, PROTEAN might explain that KSAR55 is feasible because:

```
Did-include helix6-2 in pa2.
Has helix6-2 anchoring constraint.
```

Similarly, it might explain that it desires to:

```
Quickly do-position long constraining secondary-structure
        in pa2 with strong constraint.
```

until it achieves a state in which:

```
Has pa2 status complete.
```

Fourth, given a framework, each control decision's description serves both as a machine-interpretable representation for control reasoning and as a human-interpretable representation for explanation. We can make a stronger claim that a system explains its behavior in terms of its own understanding of that behavior.

Insert Figure 43

4.4 Learning in BB*

A system's ability to learn depends upon several factors, including the following: (a) the power of the system's learning procedures; (b) the quality of the data to which it applies those procedures; and (c) the depth and organization of the system's knowledge about relevant concepts. A framework improves each of these factors. As a consequence, it improves both the efficiency and the accuracy of learning.

Let us reconsider MARCK's effort to learn the heuristic, "Prefer-Anchoring-over-Yoking."

Figure 44 illustrates MARCK's behavior for the ACCORD implementation of PROTEAN. (Figure 23 in section 2 illustrates MARCK's behavior for the BB1 implementation of PROTEAN.)

At this point, PROTEAN is operating under the focus:

```
Position long rigid constraining secondary-structure
          in pa1 with strong constraint.
```

Given this focus, PROTEAN chooses to perform the action of KSAR56:

```
KSAR56: Do-yoke helix6-2 with helix4-2 in pa2 with NOE9.
```

However, the domain expert prefers the action of KSAR55:

```
KSAR55: Do-anchor helix6-2 to helix3-2 in pa2 with NOE8.
```

As discussed in section 2, the domain expert's action triggers MARCK, a learning knowledge source that tries to identify the key difference between the two KSARs and automatically program a corresponding control heuristic.

Insert Figure 44

A framework substantially reduces the number of potential differences between the two KSARs that MARCK (and the domain expert) must investigate. Given the BB1 representation, MARCK must search for differences on all attributes with the same name. Then it must ask the domain expert to choose the key attribute from among all whose values differ. Given a framework representation, MARCK can focus on corresponding parameters in corresponding patterns in the two KSARs. In this case, there are only four corresponding parameters in the two action patterns and only two of them have different values: action-keyword and constraint.

A framework also prevents MARCK from making specious comparisons. Working within BB1 knowledge structures, different or undisciplined system builders may give the same name to unrelated attributes in different knowledge sources. MARCK must pursue differences in the values of these attributes as though they were meaningful. Conversely, if different attributes happen to exhibit the same difference in values, MARCK must ask the domain expert which is the key attribute. Since the domain expert is not a programming expert and ordinarily would not appreciate the actual differences between two attributes having the same values, he or she may choose the wrong one. By contrast, a framework focuses MARCK's and the domain expert's attention on key task-specific control parameters by enforcing consistent and semantically valid naming conventions and explicitly identifying corresponding parameters. As a consequence, MARCK pursues only meaningful differences.

A framework also enhances MARCK's ability to identify the heuristic function underlying a domain expert's preference for one value of a parameter over another. MARCK inspects the knowledge base to determine whether any known modifiers favor the expert's preferred value over the system's preferred value. For example, in PROTEAN *quickly* is a defined modifier for position, which is the superordinate of anchor and yoke. In Figure 44, MARCK determines that the modifier, quickly, favors anchor over yoke, hypothesizes that this is the key difference between the two KSARs, and asks the domain expert for confirmation. If the modifier, quickly, were not already defined, MARCK would search for the key attribute of the identified parameter and for an appropriate canonical function, automatically program a new heuristic function as discussed in section 2, and record it in the knowledge base as the definition of a new modifier for the concept, position.

A framework enables MARCK to introduce a new heuristic at the appropriate level of the control plan. Thus, once MARCK identifies quickly as the key modifier, it can search the control plan for the highest superordinate of its current focus that specifies position or one of its subordinates in the type hierarchy as the action keyword. With confirmation from the domain expert, MARCK inserts the new modifier at that level of the plan. If the expert objected, MARCK could work down the plan searching for the appropriate level at which to insert the new modifier.

Finally, MARCK no longer needs its Lisp-English translator since all of the objects on which it operates are already expressed in ACCORD's stylized English representation. Thus, MARCK completes its learning by simply inserting the new modifier before the corresponding parameter in its focus decision on the blackboard and in the control knowledge sources that generate that decision.

```
Quickly do-position a long rigid constraining
        secondary-structure in pa1 with
        strong constraint.
```

These advantages apply to other learning procedures as well. For example, as discussed in section 2, we have been working on a set of knowledge sources called WATCH to form inductive generalizations of sequences of executed actions. For example, suppose a domain expert executes the following sequence of actions:

```
Anchor Helix2-1 to Helix1-1 in PA1 with NOE15.
Anchor Helix3-1 to Helix1-1 in PA1 with NOE19.
```

The WATCH knowledge sources would determine that:

```
Helix2-1 $is-a Helix.
Helix3-1 $is-a Helix.
```

```
Long Helix2-1 = 90.
Long Helix3-1 = 70.
NOE15 $is-a NOE.
NOE16 $is-a NOE.
```

and hypothesize that the domain expert's current focus is to:

```
Perform:
        Anchor Long Helix to Helix1-1 in PA1 with NOE.
```

In principle, any BB1 system could provide the data required for inductive generalization. In practice, however, such learning ordinarily is not feasible for systems implemented directly in BB1 knowledge structures. Given the unrestricted number of KSAR attributes, the space of possible generalizations is intractably large. Moreover, given an undisciplined approach to attribute naming, the learning data are liable to be extremely noisy. They may support specious generalizations, while entirely concealing valid generalizations. By contrast, a framework such as ACCORD vastly reduces the space of possible inductions and guarantees that it is internally consistent, unambiguous, and semantically valid. Given a framework, we believe that inductive generalization is feasible for BB1 systems.

5. Knowledge Engineering within the BB* Environment

The BB* environment facilitates knowledge engineering in two general ways. First, the task-specific, domain-independent knowledge embodied in a framework facilitates the design and implementation of new applications. To illustrate this potential, we discuss our experience in building a prototype of the SIGHTPLAN system [50] within BB1-ACCORD. We then consider the space of domains in which arrangement problems occur and ACCORD's applicability in different regions of that space. Second, the BB* environment's capability for open systems integration facilitates the development of *multi-faceted systems* that perform a greater variety of tasks than conventional knowledge-based systems. To illustrate this potential, we discuss two hypothetical multi-faceted systems, an expert arrangement-assembler and an expert protein-analyzer.

5.1 Building SIGHTPLAN: A New Application of BB1-ACCORD

5.1.1 SIGHTPLAN's Problem

SIGHTPLAN must arrange pieces of construction equipment (e.g., cranes and trailers) and construction areas (e.g., access roads and lay-down areas) in a two-dimensional construction site to satisfy a variety of constraints. Part-whole relations exist among some of these objects (e.g., the employee-facilities include some trailors and a rest area). Part-whole relations also exist among sub-regions of the construction site (e.g., the building-zone includes the building-site and all of its borders). Available constraints include object-based constraints (e.g., the rest area must be within a short distance of the trailers) and context-based constraints (e.g., the access road must intersect the perimeter of the construction site on two sides). Since construction projects proceed in identifiable stages, the layout design must include sub-layouts for different stages. Further, there are transitional constraints between the stages (e.g., the crane must move from the northwest corner of the building site to the southeast corner of the building site between stages 1 and 2). (See [50] for a more detailed description of the problem of designing construction-site layouts.)

Despite the obvious dissimilarities between proteins and construction sites, the problem of designing a construction site closely resembles the problem of modeling the construction of a protein. In both cases, the problem-solver must arrange physical objects in a spatial context to satisfy constraints. It must accommodate a variety of constraints, including part-whole relations, objects-based constraints, and context-based constraints. It must design multiple-component solutions for different time intervals and provide legal transitions from each component solution to its successor.

On the other hand, while the two problems have formal similarities, SIGHTPLAN's problem is substantially less complex than PROTEAN's problem. SIGHTPLAN must deal with tens or hundreds of objects, while PROTEAN must deal with hundreds or thousands of objects. SIGHTPLAN must arrange objects in a two dimensional space, while PROTEAN must arrange objects in a three-dimensional space. SIGHTPLAN must design layouts that incorporate fewer than ten discrete states, while PROTEAN must construct proteins that move through a continuous family of conformations. SIGHTPLAN knows in advance how many stages it must consider and which objects and constraints belong in each state, while PROTEAN must identify protein states and their constituent objects and constraints as part of its reasoning process. SIGHTPLAN must design a small number of satisfactory site layouts, while PROTEAN must construct the entire family of legal protein structures.

Because of the formal similarities between SIGHTPLAN's problem and PROTEAN's problem, SIGHTPLAN's principal designers, Iris Tommelein and Ray Levitt, decided to develop it within BB1-ACCORD and we collaborated with them on a prototype system. The following sections discuss how ACCORD affected the design and implementation of different aspects of the SIGHTPLAN prototype.

5.1.2 Choosing a Method

As discussed above, a problem-solving system could, in principle, solve an arrangement problem by any of several different methods. Enumerating and characterizing alternative methods and then choosing and operationalizing an appropriate method for a particular application are time-consuming processes that can determine the success or failure of a system-building effort. For example, it took approximately one person-year of effort to consider alternative methods for PROTEAN and to operationalize the chosen assembly method.

The very existence of a relevant framework can facilitate this process by suggesting a candidate method in a clearly operational form. If the framework already has been applied in other domains, information about those applications can facilitate evaluation of the method for the new application. Thus, it took approximately one person-month for Tommelein and Levitt to decide to use the assembly method for SIGHTPLAN.

5.1.3 Basic Knowledge Acquisition

Knowledge acquisition requires a conceptual analysis of the knowledge required by an application and a technical analysis of appropriate knowledge representation structures. For example, knowledge acquisition for PROTEAN began with unstructured discussions with domain experts to discover the important domain concepts. The initial PROTEAN knowledge base was an unprincipled collection of Lisp functions and data structures, converted to its current declarative form during a reimplementation phase. All stages of knowledge acquisition required close collaboration between domain experts and knowledge engineers.

A framework can facilitate knowledge acquisition by capturing the conceptual analysis common to a class of applications, identifying appropriate knowledge representation structures, and providing a software environment in which to build the new knowledge base. For example, ACCORD requires domain-specific extensions of its conceptual network branches representing objects, contexts, and constraints and specification of low-level functions for anchoring, yoking, appending, etc. Thus, knowledge acquisition for SIGHTPLAN began directly with the introduction of particular objects, contexts, and constraints into ACCORD's skeletal concept network and investigation of alternative approaches to building low-level functions. In addition, domain experts were able to do much of the knowledge acquisition, with modest amounts of assistance from a knowledge engineer. Of course, since the framework provides much of the actual code necessary to represent the knowledge, there is a substantial reduction in the number of lines of new code generated during knowledge acquisition.

5.1.4 Domain Knowledge Sources

A framework's action hierarchy guides the design of domain knowledge sources. Basically, the system builder should consider designing one or more knowledge sources to instantiate each terminal action type. The hierarchical classification of action types provides a nice organization of the knowledge sources and the sequence in which to develop them. Further, the knowledge sources developed for previous applications can provide valuable prototypes for new applications.

Without the benefit of ACCORD, the first version of PROTEAN had knowledge sources for anchoring and yoking, which it used to position structures within one complete arrangement. After studying the performance of this system, it became apparent that PROTEAN needed a knowledge source for appending and only much later did it become apparent that PROTEAN needed knowledge sources for defining partial arrangements. (PROTEAN still does not have knowledge sources for integrating partial arrangements and coordinating them at multiple levels of abstraction.) Each knowledge source, especially the early ones, required a significant design effort and each successive one had to be coordinated with those developed so far. Since we did not anticipate all contexts in which knowledge sources might interact, we repeatedly modified previously implemented knowledge sources to disambiguate the relationships among them.

By contrast, SIGHTPLAN's current domain knowledge sources are close translations of PROTEAN's domain knowledge sources and were implemented in a matter of days. Although we anticipate that SIGHTPLAN and PROTEAN eventually will have many distinct knowledge sources, we expect the translated knowledge sources to endure as the core of the SIGHTPLAN system. If these expectations are borne out, we will extend ACCORD and other frameworks to include a repertoire of prototype domain knowledge sources and introduce capabilities for automatically instantiating them in new domains.

5.1.5 Control Knowledge Sources

A framework facilitates the development of control knowledge sources in several ways. First, its action, event, and state templates articulate a set of candidate control concepts. Thus, PROTEAN's system builders had to discover key control parameters, such as action class, anchoree, and constraint, and appropriate modifiers, such as quickly, restricted, and strong. By contrast, SIGHTPLAN's sytem builders could begin by considering the formal parameters in ACCORD's action types as candidate control parameters and by considering the high-level concept types and conceptual modifiers in ACCORD's skeletal concept network. Second, as in the case of domain knowledge sources, some control knowledge sources transfer almost directly to applications in new domains. For example, the prototype SIGHTPLAN system uses the basic strategy that PROTEAN uses for small proteins. Of course, SIGHTPLAN introduces some new modifiers and gives many of the common modifiers new procedural definitions. In addition, we expect to develop more powerful strategies for the two systems that differ more substantially. Again, however, the opportunity to transfer some of the control knowledge permits rapid prototyping of a new application. After we have gained more experience with a range of applications, we plan to develop skeletal control knowledge sources for different subclasses and automatic methods for instantiating them in new domains. Finally, a framework's perspicuous representation makes it easy to articulate and program alternative control strategies. We plan to comparatively evaluate a variety of control strategies for both PROTEAN and SIGHTPLAN.

5.2 The Scope of ACCORD

5.2.1 Arranging Physical Objects in a Spatial Context

ACCORD naturally applies to tasks involving the arrangement of physical objects in a spatial context. PROTEAN and SIGHTPLAN are esoteric examples of such domains. However, consider, for example, the mundane task of furniture arrangement: arrange a specified set of furniture in a designated room. We can define each piece of furniture as a physical-object in the ACCORD knowledge base and the room as a context. We can identify part-whole relationships among furniture groups (e.g., the table-and-chairs includes the table and each of the chairs). We can identify part-whole relationships among areas of the room (e.g., the northern exposure includes a window area and a fireplace area). We can define object-based constraints on different pieces of furniture (e.g., each chair must be on a particular side of the table). We can define context-based constraints on the positions of particular pieces of furniture within the room (e.g., put the table near a window). Given this representation, we could use the ACCORD actions to define partial furniture arrangements, to position pieces of furniture within each partial arrangement, to refine the positions of furniture groups into the positions of their constituent pieces, and to integrate different partial furniture arrangements to form a complete room design.

5.2.2 Arranging Procedural objects in a Temporal context

We believe that ACCORD also applies to tasks involving the arrangement of procedural objects in a temporal context. For example, consider the task of travel planning: arrange a set of destinations in a designated time interval. We can define each destination as a temporal-object in the ACCORD knowledge base and the time interval as a context. We can define part-whole relationships among sets of destinations (e.g., the India destination includes destinations: Srinagar, Agra, Jaipur, Udaipur, Benares, and Darjeeling). We can define part-whole relationships among sub-intervals of the designated time interval (e.g., the spring interval includes May and June). We can define object-based constraints on the relative times targetted for particular destinations (e.g., go to India after Japan). We can define context-based constraints on the absolute times targeted for particular destinations (e.g., go to Japan in time for the cherry blossoms). Given this representation of the knowledge, we probably could use the ACCORD actions to develop partial itineraries, to order destinations within partial itineraries, to refine high-level destinations into more detailed itineraries for their constituent destinations, and to integrate different partial itineraries to form a complete itinerary. We plan to build at least one application of BB1-ACCORD involving procedural objects in temporal contexts in order to gain empirical evidence of its applicability to this important subclass of arrangement problems.

5.2.3 Arranging Symbolic Objects in a Symbolic Context

Expanding the potential scope of ACCORD even further, it may be possible to apply it to tasks involving the arrangement of general symbolic objects in general symbolic contexts. In particular, it may apply to objects and contexts that are not metric in character.

For example, consider a simplified project-management task: assign a set of project tasks among a designated set of individuals. We can define each task as a task-object in the ACCORD knowledge base and the set of individuals as a context. We can define part-whole relationships among task groups (e.g., the task of designing knowledge sources includes tasks for designing domain knowledge sources and designing control knowledge sources). We can define part-whole relationships among subsets of the individuals (e.g., the expert C programmers are John, Jim, Craig, and Bruce). We can define object-based constraints between different tasks (e.g., the tasks of defining domain and control action languages must be performed by the same individual). We can define context-based constraints on the assignments of particular tasks to individuals (e.g., the geometry system must be implemented by expert C programmers). Given this representation, we might be able to use the ACCORD actions to develop partial project plans, to assign tasks to individuals within partial plans, to refine the assignment of high-level tasks into assignments of their component tasks, and to integrate different partial plans to form a complete project plan.

Of course, most project-planning tasks also have a temporal dimension with associated constraints. Assuming that ACCORD applies to tasks involving the arrangement of procedural objects in a temporal context, it might be possible to apply it to the complete project-planning task: assign a set of project tasks to a designated set of individuals for completion at particular times.

5.3 Building Multi-Faceted Systems

As discussed in section 1, we require that all modules within a level of the BB* environment satisfy uniform standards of knowledge content and representation. In adhering to this design principle, we aim to achieve open systems integration of modules within a level. That is, we aim to support the development of systems that: (a) configure and augment arbitrary sets of existing modules; (b) eliminate redundancy in the contents of those modules; (c) organize the actions enabled by those modules in any appropriate organizational scheme; and (d) superimpose on their reasoning uniform capabilities for control, explanation, and learning.

To illustrate the capability for open systems integration, consider a new class of applications that we call *multi-faceted systems*. We define multi-faceted systems with reference to the three-dimensional space of knowledge we have identified in this paper: knowledge about different problem classes, knowledge about different problem-solving methods, and knowledge about different subject-matter domains. Most contemporary knowledge-based systems occupy a relatively small region of this space: each one knows how to solve a single class of problems by means of a single problem-solving method in a single subject-matter domain. In contrast, multi-facted systems expand their knowledge along one or more dimensions of the space: each one knows how to solve more than one class of problems or how to apply more than one problem-solving method or how to solve problems in more than one domain. Let us consider two hypothetical multi-faceted systems.

We are thinking of building an expert *arrangement-assembler*--that is, a system that knows how to apply the assembly method to arrangement problems in each of several subject-matter domains. As illustrated in Figure 45, the arrangement-assembler initially would integrate ACCORD, PROTEAN, and SIGHTPLAN by defining objects, constraints, and contexts for both construction and biochemical domains. Similarly, it would integrate PROTEAN and SIGHTPLAN knowledge sources (not shown in Figure 45), exploiting the fact that they use many identical knowledge sources that mention no domain-specific entities (see, for example, Figures 36 and 38). We would add knowledge about refining prototypes, identifying analogous problems, and measuring different aspects of problem-solving performance. Given this knowledge and some problem-solving experience, the arrangement-assembler could, for example: (a) automatically program prototype systems for new application domains; (b) transfer control knowledge amongrelated problem types; and (c) assess the effectiveness of control knowledge for particular problem types. In general, the arrangement-assembler could develop increasingly sophisticated arrangement-assembly expertise and apply its expertise to an expanding variety of arrangement problems.

Insert Figure 45

Now consider an expert *protein-analyzer*--that is, a system that knows how to assemble protein structures and how to explore such structures for interesting features. As illustrated in Figure 46, the protein-analyzer initially would integrate ACCORD, EXPLORE (a framework being developed by Russ Altman to find interesting features of structured descriptions), PROTEAN, and FEATURE (an application being developed by Altman to apply EXPLORE to protein conformations). Thus, it would define actions, events, and states for both the exploration and assembly methods. It would merge all biochemistry knowledge into a single type hierarchy. Similarly, it would incorporate the entire set of PROTEAN and FEATURE knowledge sources (not shown in Figure 45). We could add knowledge about controlling the various actions enabled by PROTEAN and FEATURE for particular purposes. Given this knowledge, the protein-analyzer could, for example: (a) solve a test protein and then examine the hypothesized conformations for interesting features; or (b) search for interesting features while solving a protein and pursue only hypothesized conformations that exhibit interesting features. In general, the protein-analyzer could combine different kinds of expertise to solve a variety of more complex problems.

Insert Figure 45

As these examples illustrate, BB*'s capability for open systems integration introduces the possibility of incrementally extending the depth and variety of knowledge within a single system to encompass new problem classes, problem-solving methods, and subject-matter domains. At the same time, the underlying knowledge base remains perspicuous, well-structured, and non-redundant. Finally, the system continues to employ uniform methods for control, explanation, and learning, thereby presenting a coherent face for the system as a whole.

6. The BB* Environment: Status and Plans

6.1 The BB1 Architecture

6.1.1 Generality

We put forth BB1 as a general architecture for intelligent systems. Table 1 (see section 1) briefly describes some of the application systems currently implemented or being implemented in BB1. Most of these applications are being developed by other scientists at Stanford and other research laboratories. In addition, we have shown elsewhere [23] that BB1 provides a natural architecture for the knowledge and control strategies of the Hearsay-II [12] speech-understanding system, the HASP [42] signal-interpretation system, and the OPM [26] task-planning system. The number, variety, and significance of these applications suggest that BB1 provides a generally useful architecture. As we and other scientists develop and classify new applications, we will identify empirical bounds on BB1's generality and utility.

6.1.2 Control, Explanation, and Learning

We continue to extend and improve BB1's basic capabilities for control, explanation, and learning.

In the area of control, BB1 currently has three sets of generic control knowledge sources. One set of knowledge sources (illustrated in section 2) refines an application-specific strategy by successively posting the names of control knowledge sources that post its prescribed sequences of subordinates. We also are working on a set of knowledge sources that refine a strategy expressed in framework knowledge structures by successively replacing its parameter phrases with alternative legal values (discussed in section 4). A third set of knowledge sources performs goal-directed reasoning by posting focus decisions that favor KSARs whose actions would satisfy the preconditions of other high-priority KSARs [33]. All of these generic control knowledge sources can work together, along with application-specific control knowledge sources, to construct fully integrated control plans.

In the area of explanation, BB1 currently provides demand-driven, incrementally more elaborate descriptions of a system's strategic plan, as illustrated in sections 2 and 4. In addition, BB1 can display its explanations graphically. We are investigating other more focused styles of explanation, as well as extensions of all styles to encompass a system's strategic goals and the enabling conditions of pending KSARs.

In the area of learning, BB1 currently provides the MARCK knowledge sources for learning new control heuristics from user intervention (see sections 2 and 4). We also have developed the WATCH knowledge sources for drawing inductive generalizations from domain experts' problem-solving actions. We have not yet developed the WATCH knowledge sources that automatically program new control knowledge sources to regenerate inductively acquired strategies during subsequent problem solving episodes. We also are investigating prototype instantiation and learning by analogy as methods for learning how to use general knowledge in a new domain and for transferring control knowledge among related applications.

In addition to these new developments, we are conducting experiments to evaluate the cost/benefit tradeoffs of exploiting BB1's capabilities for control, explanation, and learning.

6.1.3 Framework-Interpreter and Related Functions

We are busy implementing all framework-related functionality described in the paper. This section describes the state of the implementation as of August, 1986. However, the reader may assume that the implementation as of the current date has proceeded further. Again, the framework-interpreter is entirely independent of ACCORD and can be applied to any user-specified framework specified with the appropriate BB1 knowledge structures. Moreover, since all extensions to BB1 are designed to accommodate systems that integrate BB1 and framework knowledge structures, current application systems can exploit all aspects of the evolving implementation. We summarize the state of the implementation as follows:

- We have implemented all *$links* (e.g., *$is-a, $plays, $includes, $entails, $triggers, $enables, $causes, $promotes.*

- We have implemented the matcher, the quantifier, and the translator, but not the generator.

- We have extended the BB1 scheduler to use the matcher and quantifier to rate pending KSARs on the agenda.

- We have extended the BB1 interpreter to use the translator to convert KSAR actions exressed in framework knowledge structures into BB1 knowledge structures prior to execution.

- We are working on extensions to the agenda-maintainer to apply appropriate framework-interpretation functions during knowledge source triggering, context binding, and precondition evaluation.

- We are investigating a number of strategies that exploit the conceptual network for efficient use of framework-interpretation procedures. For example, we plan to exploit the natural discrimination networks entailed in root verb hierarchies for efficiency during invocation of knowledge sources that share common triggering, context, or precondition patterns. As a second example, we plan to exploit the known relations between previous events and the states they promote to restrict the potentially explosive search required to instantiate arbitrary state patterns.

- Finally, although the template grammar underlying our framework-interpretation procedures satisfies the requirements of current applications, we anticipate that it will prove too restrictive for later versions of these applications and for new applications. Therefore, we expect to replace it with a more powerful grammar at some time in the future.

6.2 Current and Planned Frameworks

At this time, two frameworks are implemented in BB1: ACCORD and EXPLORE.

As discussed in this paper, ACCORD embodies knowledge about assembling arrangements of objects under constraints. We have demonstrated ACCORD's applicability in PROTEAN's biochemistry domain and in SIGHTPLAN's construction domain. We have also examined its applicability to problems involving procedural objects in temporal contexts and, more generally, to problems involving symbolic objects in symbolic contexts. We continue to extend and refine the knowledge in ACCORD as our understanding of specific applications grows.

EXPLORE [2], which is being developed by Russ Altman, embodies knowledge about identifying interesting features of structured descriptions. At this time, EXPLORE is in a more preliminary stage of development than ACCORD and it is being used in only one application, Altman's FEATURE [3] system for identifying biochemically interesting features of observed and hypothesized protein structures. Like ACCORD, however, EXPLORE has potential applications in other subject-matter domains.

Finally, we plan to develop new frameworks for several tasks, including: BB1's control, explanation, and learning tasks; and the several tasks--situation assessment, planning, plan monitoring, situation simulation, and plan modification--involved in real-time process control.

In general, as we and other scientists attempt to design new frameworks within BB1 and new applications within particular frameworks, we will increase our understanding of empirical bounds on: (a) the availability and utility of knowledge at this level; (b) the range of applicability of individual frameworks; and (c) the range of frameworks BB1 can accommodate.

6.3 A New Hierarchical Level: Shells

As discussed in section 1, architecture, framework, and application represent three discrete levels on what is probably a continuum of knowledge abstractions. We plan to introduce a fourth level, *shells*. Each shell will specialize a particular framework by augmenting its task-specific language with prototypical domain and control knowledge sources that are appropriate for a particular subset of tasks.

Like Clancey's Heracles system for heuristic classification [7] and Chandrasekaran's "tools for generic tasks" [5], these shells will articulate useful control strategies for solving particular classes of problems. For example, given our experience with SIGHTPLAN, we are building an ACCORD shell that captures a domain-independent form of the knowledge sources PROTEAN uses for small proteins. We believe that they will prove useful in other domains where problems involve a relatively small number of objects and constraints. Similarly, we might develop shells for arrangement-assembly tasks in domains involving physical versus temporal objects or for domains whose contexts involve nominal versus metric dimensions.

Shells will offer an incremental advantage over frameworks in the ease of developing new applications. The system builder has only to instantiate the skeletal branches of the concept network and, perhaps, the prototypical knowledge sources that require domain-specific information. As mentioned above, we are investigating automatic prototype-instantiation capabilities to relieve the system builder of the task of instantiating knowledge sources. Of course, the system builder pays for this advantage in loss of flexibility in the reasoning process.

Our shells will differ from systems such as Clancey's and Chandrasekaran's, however, in three ways. First, they will articulate control knowledge, rather than control procedures. As a consequence, a shell may support applications that exploit any of BB1's capabilities for control reasoning, ranging from systems that apply systematic control procedures to those that reason extensively about problem-solving strategy. In addition, they can exploit this knowledge for other purposes. Second, we do not presume that there is a single correct strategy for a given task. Thus, for example, there may exist several shells for arrangement-assembly tasks with different characteristics. Third, our shells will exist in the context of the BB* environment. As a consequence, they can be configured with any other modules from the environment to form more complex, but fully integrated systems, with BB1's general capabilities for control, explanation, and learning superimposed upon them.

7. Major Results

Our major results reinforce and manifest the four themes of the paper (see Figure 1 in section 1):

- that an intelligent system reasons about its actions;

- that a system must have knowledge of its actions;

- that knowledge should be represented in an abstraction hierarchy;

- that knowledge modules within a level should satisfy uniform standards of content and representation.

We have developed the BB1 architecture for systems that reason about their situations, their goals, and their actions. BB1 systems integrate strategic and opportunistic methods to decide which goals to pursue and which actions to perform. They explain how their actions serve their goals and they learn from experience which actions help them to achieve their goals. BB1 systems reason in these several ways by dynamically constructing, modifying, executing, explaining, and learning about explicit plans for their own actions in real time.

We have empowered these systems with the generic knowledge in BB1, the task-specific knowledge in frameworks such as ACCORD, and the more specific knowledge in applications such as PROTEAN. As a consequence, these systems know what facts and states obtain in particular contexts. They know what events and states they seek. They know what actions they can perform, what events and states are necessary to enable their actions, and what events and states their actions will produce. They use their knowledge to perform the control, explanation, and learning functions required of them. Since they represent all of these different kinds of knowledge explicitly, improving or extending their performance is a matter of improving or extending their knowledge.

We have organized existing modules in the hierarchically layered BB* environment: The BB1 architecture supports multiple frameworks, each of which supports multiple applications. This organization enables us to understand and describe BB*, but more importantly, to apply and extend it. We apply BB* by building new systems that incorporate and augment existing knowledge modules, possibly exhibiting synergistic effects of independently constructed modules. We extend BB* by constructing new knowledge modules or expanding existing modules. Existing high-level modules guide and discipline the construction of subordinate modules. Low-level modules substantiate superordinate modules and suggest new opportunities for abstracting superordinate modules. Some of these extensions can be made automatically.

Finally, we have adhered to uniform standards of knowledge content and representation in constructing modules at a given BB* level. We offer a single architecture, BB1, and its associated frame-based network of knowledge structures for representing actions, events, states, and facts. Frameworks such as ACCORD must specify task-specific knowledge about actions, events, states, and facts within a representation combining: a frame-based conceptual network, linguistic templates, partial match tables, and template translations. Applications such as PROTEAN must instantiate skeletal branches of the conceptual network and specify knowledge sources that instantiate particular problem-solving actions, events, and states. As a consequence of this within-level uniformity, BB* provides open systems integration. We can configure any existing knowledge modules within any appropriate strategic paradigm to attack new problems. Moreover, we can incrementally extend the knowledge within a given system to encompass additional problem classes, problem-solving methods, or subject-matter domains. At any stage in the system's evolution, we can superimpose upon it higher-level generic knowledge about control, explanation, and learning to produce a fully integrated and coherent face for the system as a whole.

From an engineering perspective, BB* may be viewed as a layered computing environment. BB1 constitutes a general-purpose "virtual computer" for programs that articulate and reason about their own actions. It offers a data representation and instruction set of considerable generality. Frameworks such as ACCORD constitute higher-level programming languages. They provide the more complex data representations and macro operators relevant in narrower, but still significant, sets of programs. Applications such as PROTEAN constitute individual programs developed within the environment. They can be programmed in the "machine language" of BB1 or in the higher-level language of an appropriate framework. Like higher-level languages in conventional computing environments, frameworks harness the power of BB1, enabling applications builders to write better programs more easily. BB* differs most from conventional computing environments in its orientation toward intelligent systems: programs that perform knowledge-intensive reasoning about the problems they solve and about their own problem-solving behavior.

From a scientific perspective, BB* may be viewed as an elementary theory of intelligent systems. Like all scientific theory, theories of intelligence carry an inevitable tension between generality and power. Efforts to design encompassing architectures strive for generality: to formulate fundamental laws of artificial intelligence. Efforts to develop task-specific frameworks (or still more specific shells) strive for power: to articulate more constraining laws for a narrower range of intelligent behavior. In both cases, effective application systems confirm predictions of the proposed theory. The BB* environment--in which the BB1 architecture supports multiple frameworks and each framework supports a range of specific shells and applications--constitutes a theoretical paradigm in which we can realize both generality and power.

References

[1] Allen, J.F.
Towards a general theory of action and time.
Artificial Intelligence 23:123-154, 1984.

[2] Altman, R.
EXPLORE.
Technical Report, Stanford University Knowledge Systems Laboratory, 1986.

[3] Altman, R.
FEATURE.
Technical Report, Stanford University Knowledge Systems Laboratory, 1986.

[4] Buchanan, B., Hayes-Roth, B., Lichtarge, O., Hewett, M., Altman, R., Rosenbloom, P., and Jardetzky.
Reasoning with symbolic constraints in expert systems.
Technical Report, Stanford, Ca.: Stanford University, 1985.

[5] Chandrasekaran, B.
Generic tasks in knowledge-based reasoning: Characterizing and designing expert systems at the right' level of abstraction.
Proceedings of the IEEE Computer Society Second International Conference on Artificial Intelligence Applications , 1985.

[6] Clancey, W.J.
Acquiring, representing, and evaluating a competence model of diagnostic strategy.
Technical Report HPP-84-2, Stanford, Ca.: Stanford University, 1984.

[7] Clancey, W.J.
Heuristic classification.
Artificial Intelligence 27:289-250, 1985.

[8] Corkill, D.D., Lesser, V.R., and Hudlicka, E.
Unifying data-directed and goal-directed control: An example and experiments.
Proceedings of the AAAI :143-147, 1982.

[9] Davis, R.
Applications of meta level knowledge to the construction, maintenance, and use of large knowledge bases.
Technical Report Memo AIM-283, Stanford University Artificial Intelligence Laboratory, 1976.

[10] Dodhiawala, R., and Jagannathan, V.
SADVISOR.
Technical Report, Boeing Computer Services, 1986.

[11] Duncan, D.
PROCHEM.
Technical Report, Stanford, Ca.: Stanford University, 1986.

[12] Erman, L.D., Hayes-Roth, F., Lesser, V.R., and Reddy, D.R.
The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty.
Computing Surveys 12:213-253, 1980.

[13] Erman, L.D., London, P.E., and Fickas, S.F.
The design and an example use of Hearsay-III.
Proceedings of the Seventh International Joint Conference on Artificial Intelligence :409-415, 1981.

[14] Feigenbaum, E.A.
The art of artificial intelligence.
Proceedings of the 5th International Joint Conference on Artificial Intelligence :1014-1029, 1977.

[15] Fikes, R.E., Hart, P.E., and Nilsson, N.J.
Learning and executing generalized robot plans.
Artificial Intelligence 3:251-288, 1972.

[16] Friedland, P.E., and Iwasaki, Y.
The concept and implementation of skeletal plans.
Technical Report, Stanford University, 1983.

[17] Genesereth, M.R., and Smith, D.E.
Meta-level architecture.
Technical Report HPP-81-6, Stanford, Ca.: Stanford University, 1982.

[18] Goos, G., and Hartmanis, J. (Eds.).
Distributed systems - Architecture and implementation.
New York: Springer-Verlag, 1981.

[19] Green, Paul E. (Ed.).
Computer network architectures and protocols.
New York: Plenum Press, 1982.

[20] Harvey, J., and Hayes-Roth, B.
WATCH: Inductive abstration of control strategies.
1986.

[21] Hasling, D.W., Clancey, W.J., and Rennels, G.
Strategic explanations for a diagnostic consultation system.
Technical Report STAN-CS-83-996, Stanford, Ca.: Stanford University, 1983.

[22] Hayes-Roth, B.
BB1: An architecture for blackboard systems that control, explain, and learn about their own behavior.
Technical Report HPP-84-16, Stanford, Ca.: Stanford University, 1984.

[23] Hayes-Roth, B.
A blackboard architecture for control.
Artificial Intelligence Journal 26:251-321, 1985.

[24] Hayes-Roth, B., and Hewett, M.
Learning Control Heuristics in a Blackboard Environment.
Technical Report HPP-85-2, Stanford, Ca.: Stanford University, 1985.

[25] Hayes-Roth, B., Buchanan, B Lichtarge, O., Hewett, M, Altman, R., Brinkley, J., Cornelius, C., Duncan, B., Jardetzky, O.
Elucidating protein structure from constraints in PROTEAN.
Technical Report KSL-85-35, Stanford, Ca.: Stanford University, 1985.

[26] Hayes-Roth, B., Hayes-Roth, F., Rosenschein, S., and Cammarata, S.
Modelling planning as an incremental, opportunistic process.
Proceedings of the International Joint Conference on Artificial Intelligence 6:375-383, 1979.

[27] Hayes-Roth, F., and Lesser, V.R.
Focus of attention in the Hearsay-II speech understanding system.
Proceedings of the Fifth International Joint Conference on Artificial Intelligence :27-35, 1977.

[28] Jagannathan, V., Baum, L., and Dodhiawala, R.
KRYPTO.
Technical Report, Boeing Computer Services, 1986.

[29] Jardetzky, O., Lane, A., Lefevre, J-F., Lichtarge, O., Hayes-Roth, B., and Buchanan, B.
Determination of macromolecular structure and dynamics by NMR.
Proceedings of the NATO Advanced Study Institue: NMR in the Life Sciences , 1985.

[30] Kitzmiller, T., and Baum, L.
RAPS.
Technical Report, Boeing Computer Services, 1986.

[31] Klahr, D., Langley, P., and Neches, R.t.
Self-modifying production system models of learning and development.
Cambridge, Ma.: Bradford Books, 1983.

[32] Lenat, D.B.
EURISKO: A program that learns new heuristics and domain concepts.
Artificial Intelligence 21:61-98, 1983.

[33] M. Vaughan Johnson.
Goal-directed reasoning in BB1.
Technical Report, Stanford, Ca.: Stanford University, 1986.

[34] MacMillan, S.
.
Technical Report, FMC Central Engineering Laboratories, 1986.

[35] McCarthy, J.
The advice taker.
In Minsky, M. (editor), *Semantic Information Processing*, . Cambridge, Ma.: MIT Press, 1968.

[36] McDermott, D.
A temporal logic for reasoning about processes and plans.
Cognitive Science 6, 1982.

[37] Miller, G.A., Galanter, E., and Pribram, D.H.
Plans and the structure of behavior.
New York: Holt, Rinehart, and Winston, Inc, 1960.

[38] Mitchell, T., Utgoff, P.E., Nudel, B., and Banerji, R.B.
Learning problem-solving heuristics through practice.
Proceedings of the International Joint Conference on Artificial Intelligence :127-134, 1981.

[39] Mostow, D.J., and Hayes-Roth, F.
Operationalizing heuristics: Some AI methods for assisted AI programming.
Proceedings of the International Joint Conference on Artificial Intelligence , 1979.

[40] Murphy, A., and Dodhiawala, R.
SIMLAB.
Technical Report, Boeing Computer Services, 1986.

[41] Murphy, A., and Jagannathan, V.
PHRED.
Technical Report, Boeing Computer Services, 1986.

[42] Nii, H.P., Feigenbaum, E.A., Anton, J.J., and Rockmore, A.J.
Signal-to-symbol transformation: HASP/SIAP case study.
AI Magazine 3:23-35, 1982.

[43] Pearson, G., and Yao, J.
Mission planning for an autonomous vehicle.
Technical Report, FMC Central Engineering Laboratoris, 1986.

[44] Reddy, R., and Newell, A.
Multiplicative speedup of systems.
In Jones, A. (editor), *Perspectives on Computer Science*, . New York: Academic Press, 1977.

[45] Rosenbloom, P.S., and Newell, A.
Learning by chunking: Summary of a task and a model.
Proceedings of the American Association for Artificial Intelligence :255-258, 1982.

[46] Sacerdoti, E.D.
Planning in a hierarchy of abstraction spaces.
Artificial Intelligence 5:115-135, 1974.

[47] Simon, H.A.
The Sciences of the Artificial.
Cambridge, Ma.: The M.I.T. Press, 1969.

[48] Sowa, J.F.
Conceptual Structures: Information Processing in Mind and Machine.
Reading, Ma.: Addison-Wesley, 1984.

[49] Tanenbaum, A.S.
Computer networks.
Engelwood Cliffs, N.J.: Prentice-Hall, Inc., 1981.

[50] Tommelein, I., Johnson, V., Levitt, R., and Hayes-Roth, B.
A prototype of the SIGHTPLAN system for the design of construction site layouts.
Technical Report Technical Report, Stanford, Ca.: Stanford University, 1986.

[51] Zimmerman, H.
A standard layer model.
In Paul E. Green (editor), *Computer network architecture and protocols*, . New York: Plenum Press, 1982.

Figure 1. Four Themes. (a) An intelligent system reasons about its actions. The BB1 architecture provides knowledge structres and a basic mechanism for control, explanation, and learning. (b) To perform effectively, a system must have knowledge about its actions. Frameworks explicitly represent knowledge about task-specific actions, events, and states and the relationships among them. (c) Knowledge is represented in an abstraction hierarchy. The BB* environment comprises an evolving body of knowledge: the BB1 architecture, various task-specific frameworks, such as ACCORD, and various domain-specific applications, such as PROTEAN (see Table 1). Conversely, an application system such as PROTEAN instantiates the knowledge structures in a framework such as ACCORD, which instantiates the knowledge structures in BB1. (d) Knowledge modules within a level satisfy uniform standards of knowledge content and representation. As a consequence, BB* achieves open systems integration: Independently constructed modules can be fully integrated in implementation and reasoning. For example, we could create an expert protein analyzer that integrates the knowledge and reasoning of two frameworks, ACCORD and EXPLORE, and two applications, PROTEAN and FEATURE, to solve more complex problems than it could solve with actions from either one alone.

(next page)

Figure 1

(caption on preceding page)

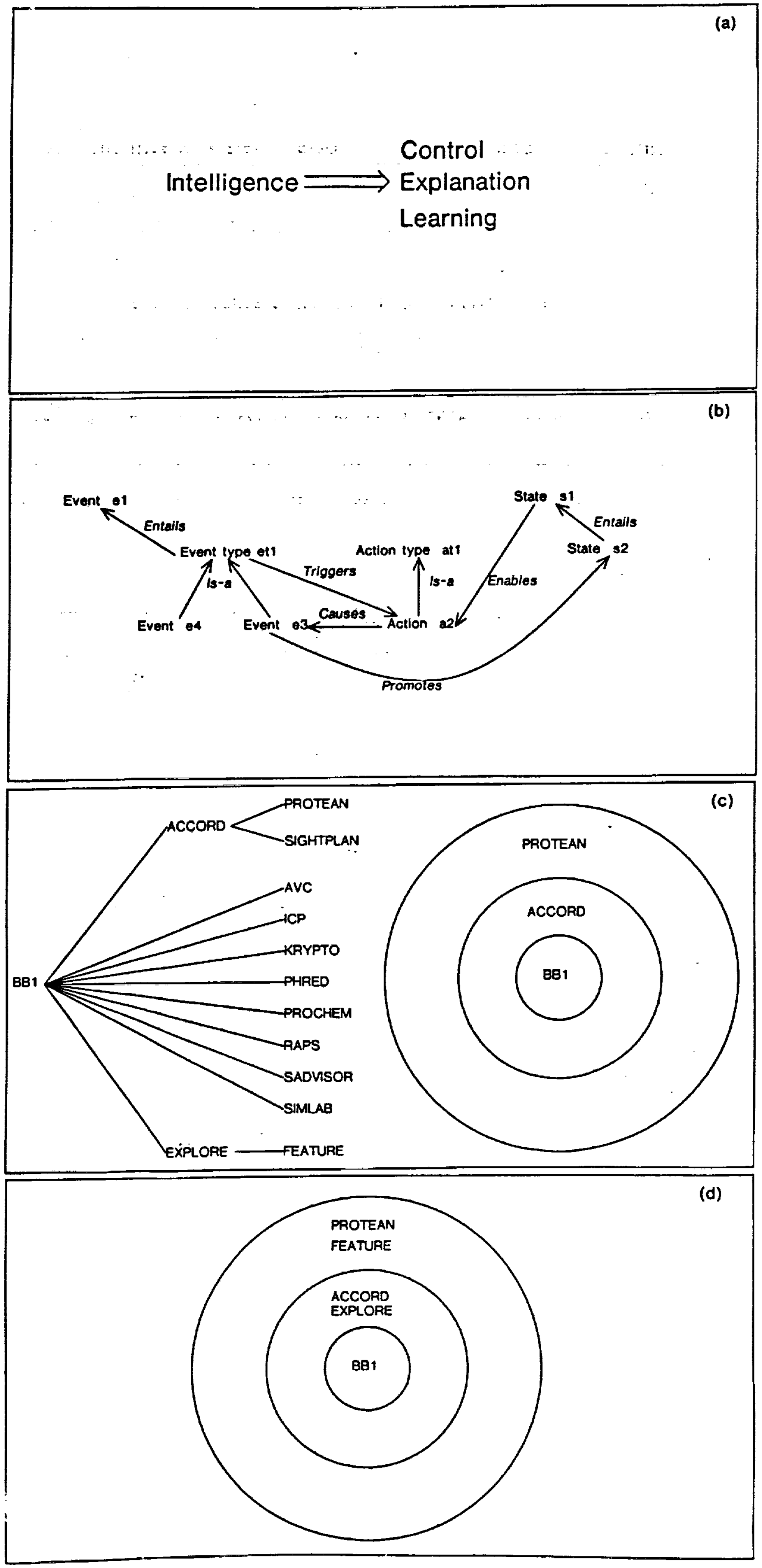
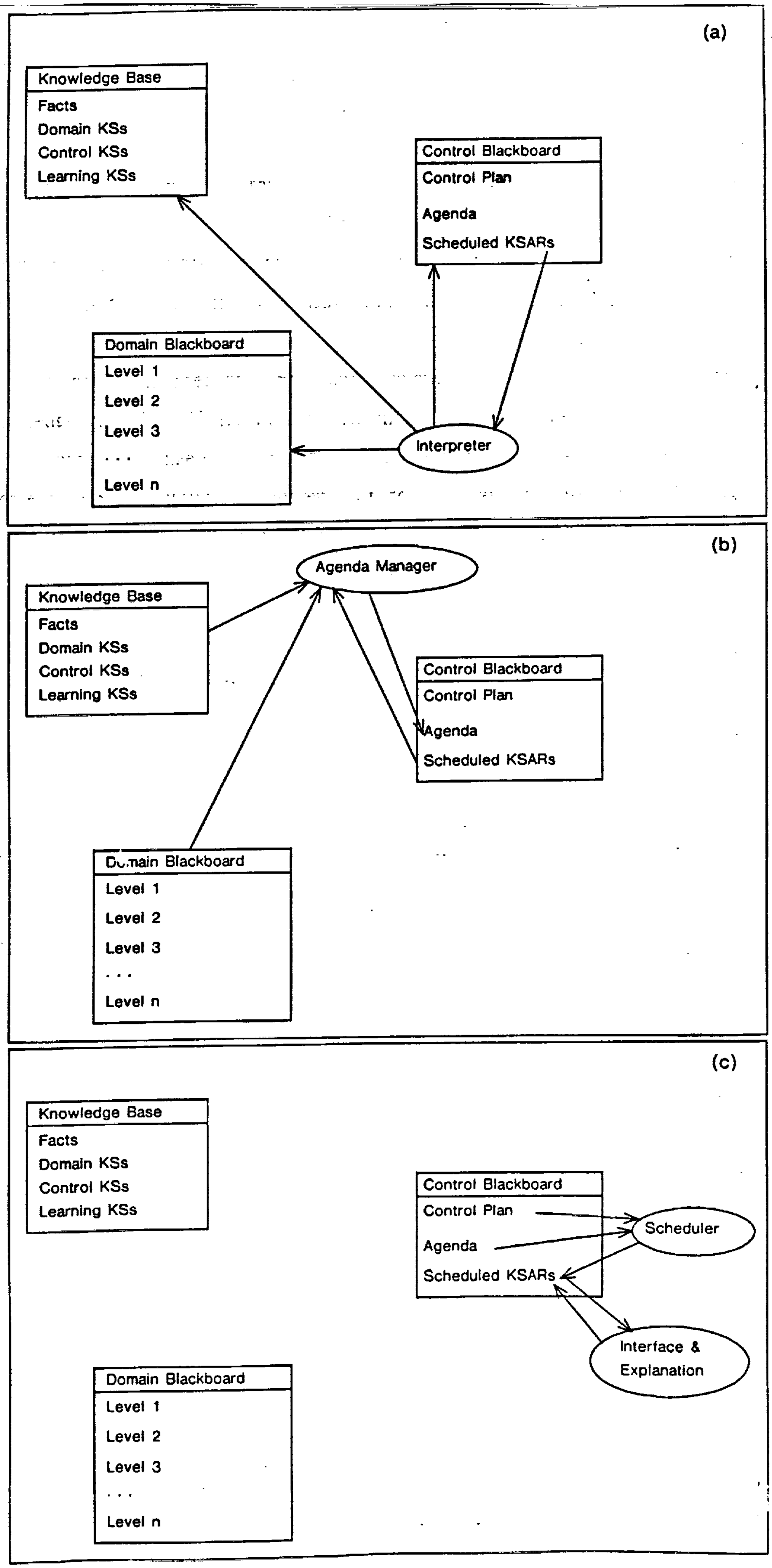


Figure 2. The BB1 Blackboard Control Architecture. Domain knowledge sources solves problems by constructing hypothetical solutions to them on the domain blackboard. Control knowledge sources construct plans for the system's actions on the control blackboard. Learning knowledge sources modify information in the knowledge base. The BB1 execution cycles comprises three steps. (a) The interpreter executes the action of the most recently scheduled KSAR (knowledge source activation record), producing changes to the contents of some blackboard or the knowledge base. (b) These blackboard changes trigger other domain, control, and learning knowledge sources. The agenda-manager adds corresponding KSARs to the agenda. (c) The scheduler rates each KSAR against the current control plan and chooses one KSAR for execution. Unless it has been instructed to operate autonomously, the scheduler also invites the user to request an explanation for the chosen action or to request other information.

(next page)

igure 2
caption on
preceding page)



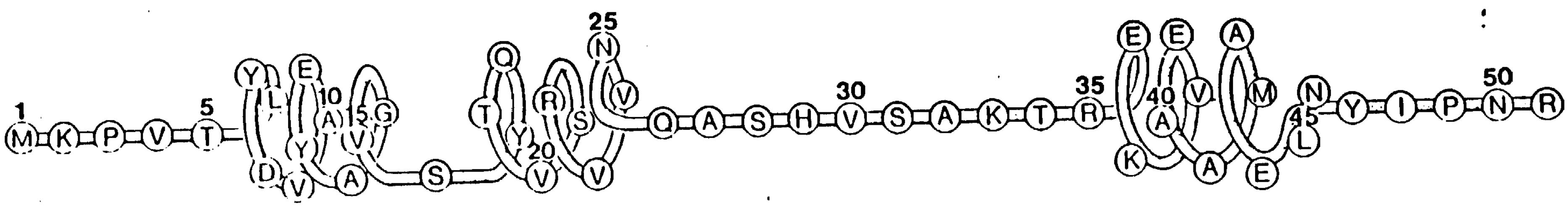


Figure 3. Primary and Secondary Structure of the Lac-Repressor Headpiece. The lac-repressor's primary structure is a unique sequence of 51 amino acids, each of which is one of the 20 unique amino acids. Its secondary structure includes three alpha-helices, each of which is defined by a series of repeated angular turns in the protein's backbone. Interspersed among its helices, the lac-repressor headpiece has random coils, segments of the primary structure that show no identifiable regularity.

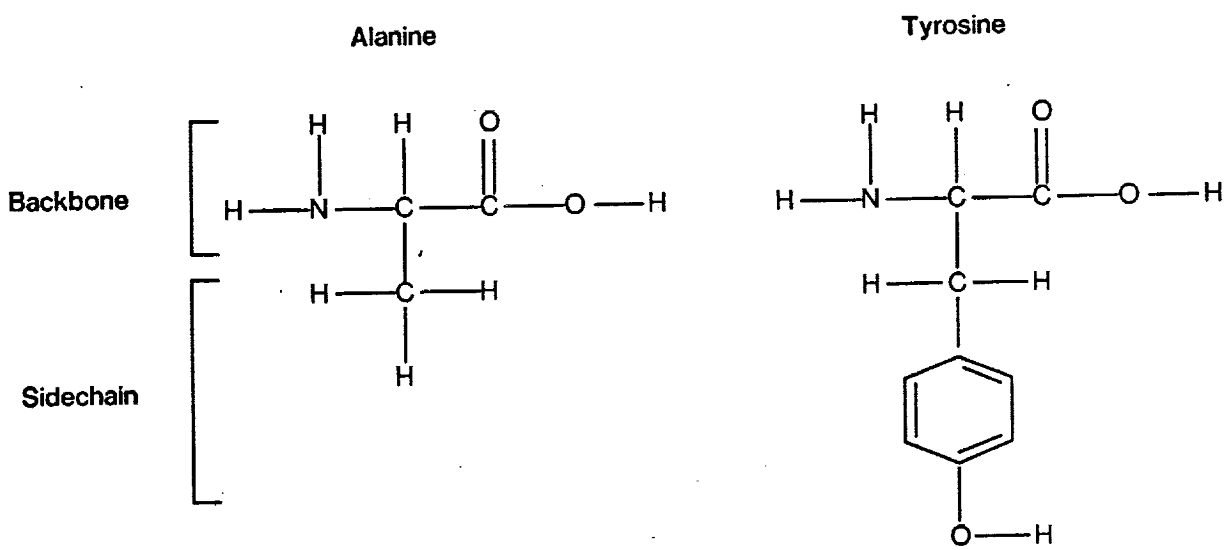


Figure 4. Two Amino Acids: Alanine and Tyrosine. As these examples illustrate, each amino acid has a common part, at which it bonds to neighboring amino acids to form the backbone of a protein, and a unique sidechain that distinguishes it from other amino acids.

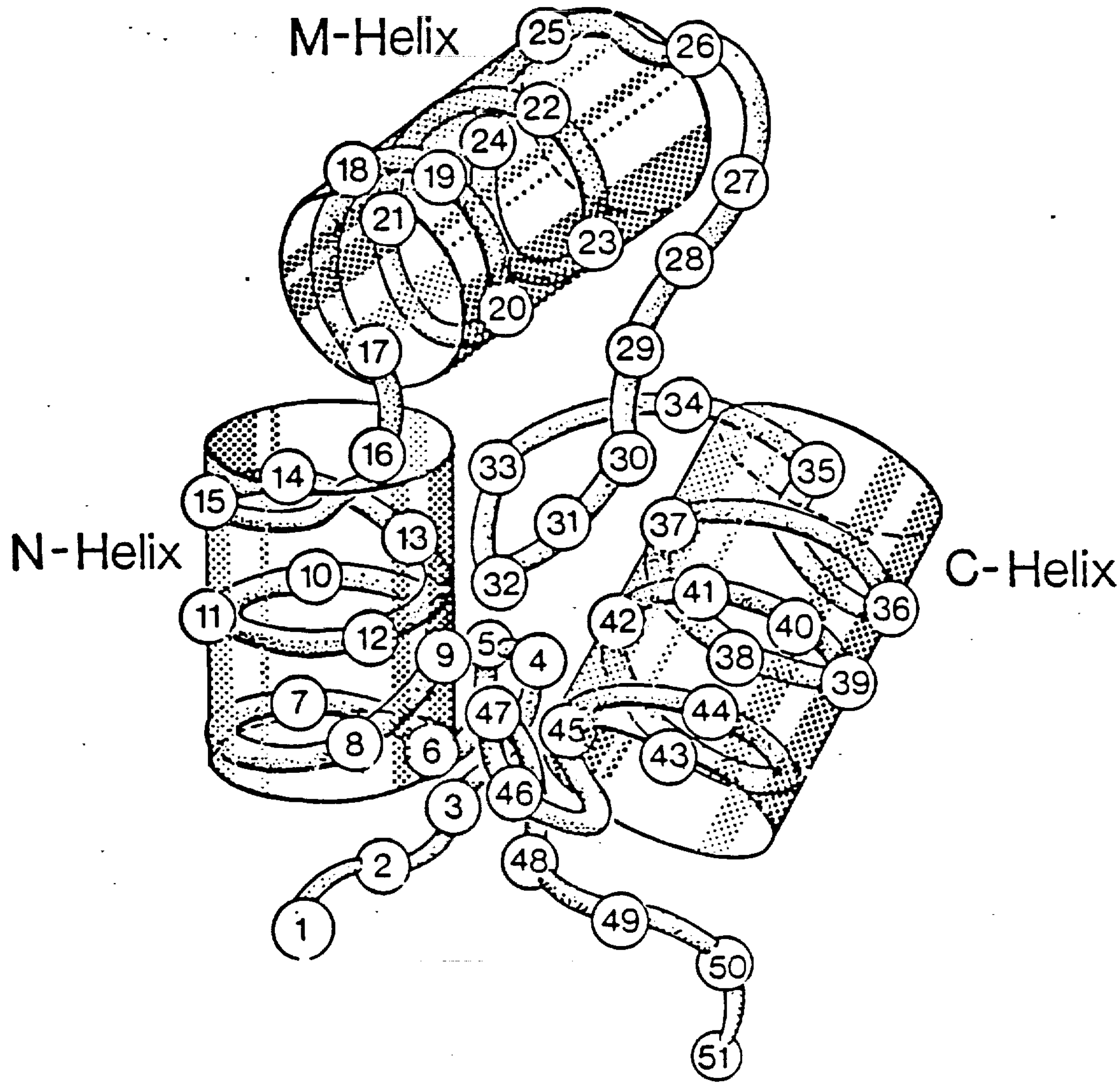


Figure 5. The Tertiary Structure of the Lac-Repressor Headpiece. The primary and secondary structure of the protein fold in three-dimensional space, packing all component structures into a globular molecule.

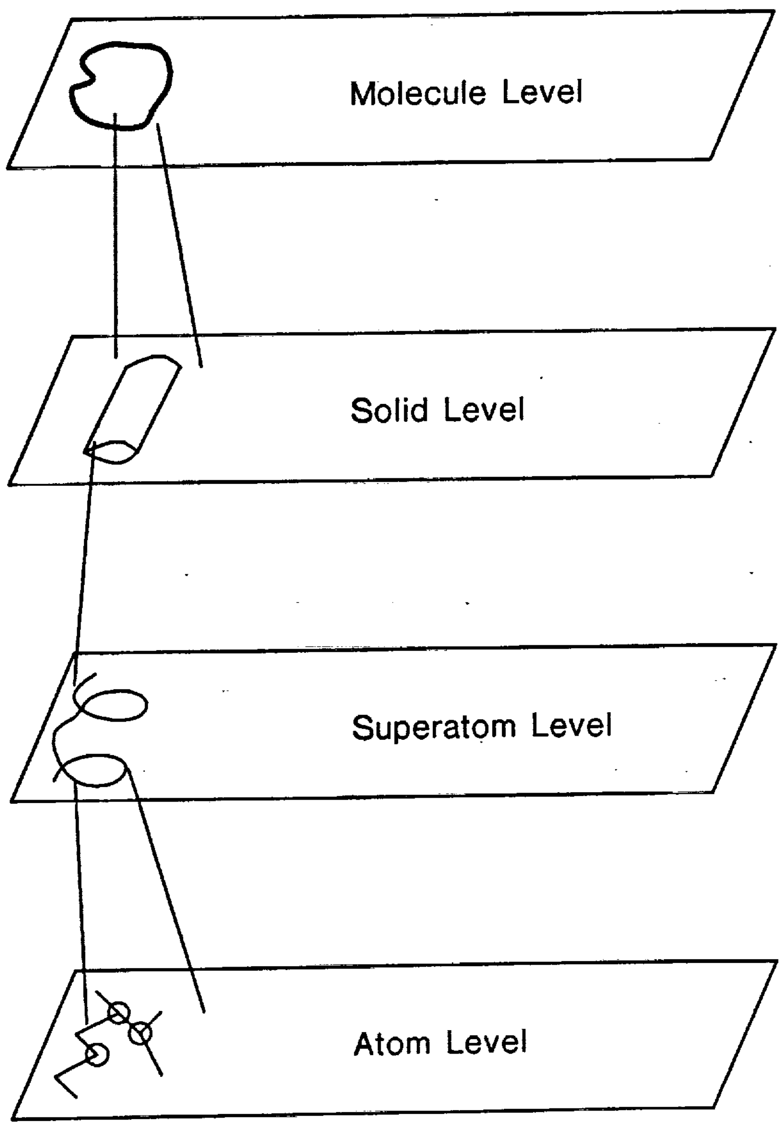
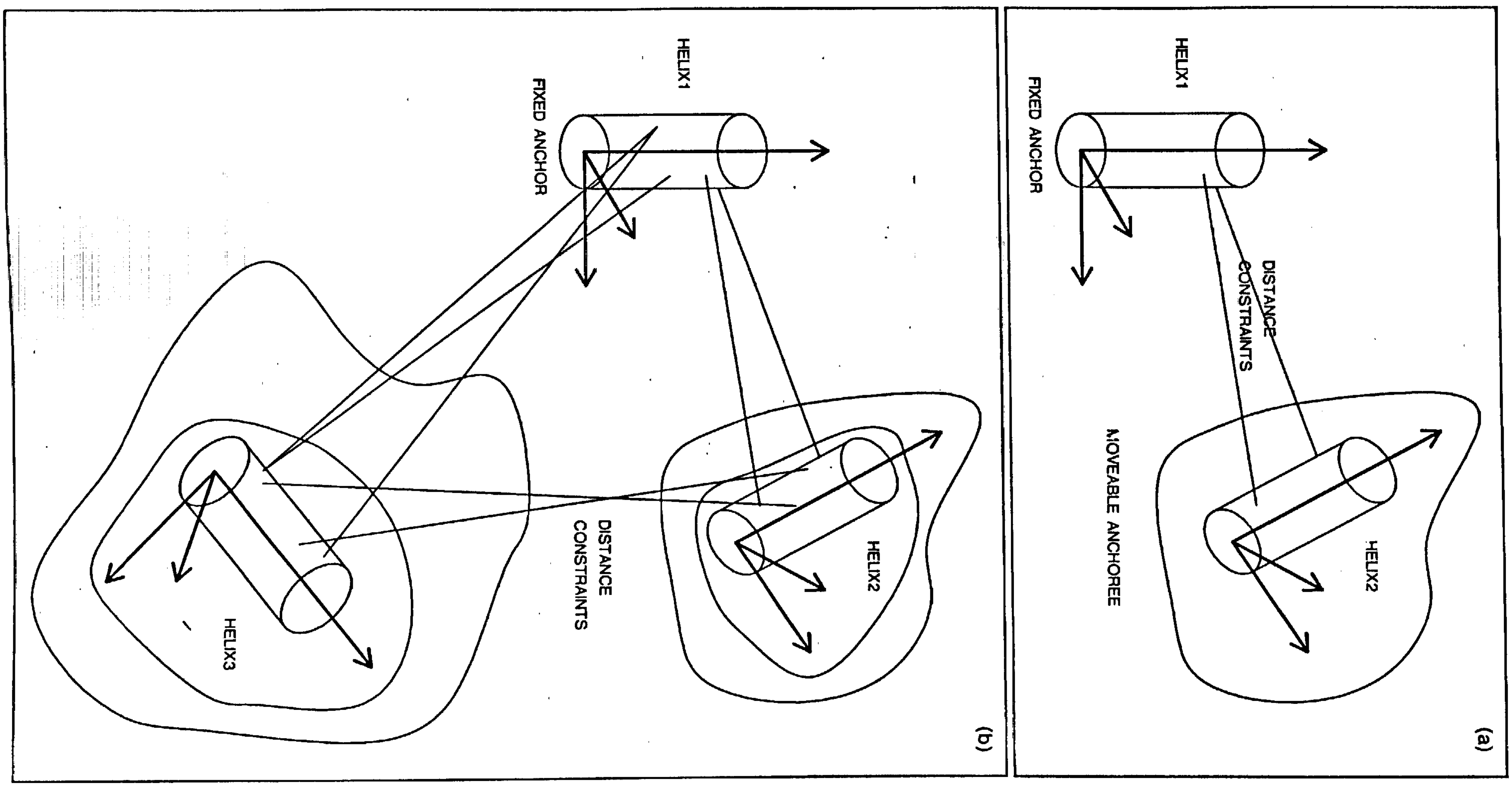


Figure 6. PROTEAN's Levels of Reasoning. At the molecule level, PROTEAN reasons about the size, shape, and density of the protein molecule. At the solid level, it reasons about the relative positions of the test protein's secondary structures, represented as geometric solids. At the superatom level, it reasons about the positions of each amino acid's constituent peptide unit and sidechain. At the atom level, it reasons about the positions of individual atoms.

Figure 7. Constraint Application in PROTEAN. (a) PROTEAN assumes a fixed position for helix1 and anchors helix2--that is, it determines that helix2 can lie in any location within the outlined region and still satisfy its constraints with helix1. In (b) PROTEAN yokes helix2 and helix3--that is, it prunes the locations previously identified for these helices to include only those that satisfy constraints between them.



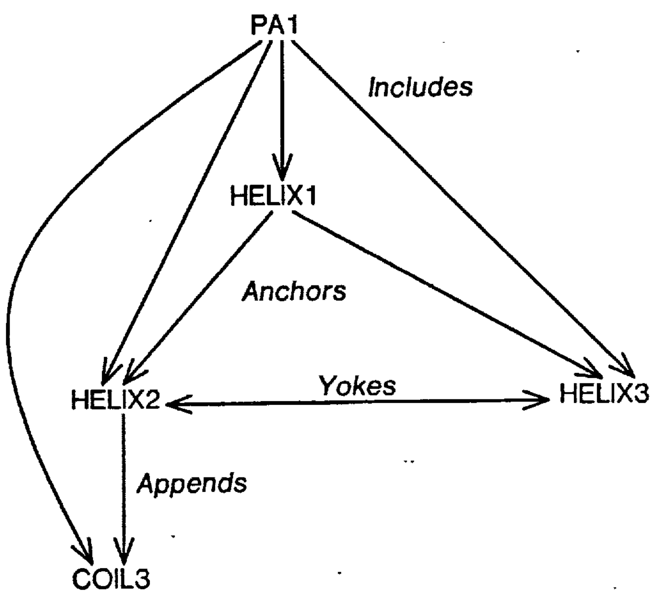


Figure 8. A Partial Solution for the Lac-repressor Headpiece. Pa1 includes helix1, helix2, helix3, and coil3. Helix1, which has been defined as the anchor of pa1, anchors helix2 and helix3. Helix2 appends coil3, which has no constraints with the anchor. Helix2 and helix3 yoke one another with the constraints between them.

```
Name - Helix1
Role - Anchor
Activated? - NIL
Type - A
Number - 1
Sequence - (LEU6 TYR7 ASP8 VAL9 ALA10 GLU11 TYR12 ALA13 GLY14)
Number-AA - 9
Percent-AA - 0.1764706
Internal-Constraints - NIL
Number-Internal-Constraints - 0
External-Constraints - (3 4 5 6 7 8 9 10 11 12)
Number-External-Constraints - 10
Parameters - ((ORIGIN 0 0 0)
              (REFERENCE 2.3 0 0)
              (TIP 0 0 13.5))
Constraints-To-Other-Structures - ((RANDOMCOIL1 0 NIL)
                                   (RANDOMCOIL2 0 NIL)
                                   (HELIX2 2 (3 7))
                                   (RANDOMCOIL3 1 (8))
                                   (HELIX3 5 (4 9 10 11 12))
                                   (RANDOMCOIL4 2 (5 6)))
```

Figure 9. An Object at PROTEAN's Solid Level. Helix1 is an anchor and it is the first alpha-helix in the lac-repressor primary sequence. It encompasses nine amino acids, numbers 6-14, approximately 18% of the entire primary structure. Helix1 has no internal constraints, but it has ten constraints to other structures, constraints numbered 3-12. Helix1's parameters describe the cylinder used to model it at the solid level of the blackboard. Its constraints-to-other-structures indicate that its ten constraints involve helices 2 and 3 and random coils 3 and 4.

Figure 10. A PROTEAN Knowledge Source: Yoke-Structures. Yoke-Structures is triggered by modification of the applied-constraints attribute of any object at the solid level. It generates a KSAR for each blackboard context in which two other structures, called anchoree1 and anchoree2, have constraints both with each other and with the triggering structure, called ps-anchor. However, a given KSAR cannot be executed until both anchorees have yoking information--previously identified legal locations. When a given KSAR is executed, Yoke-Structures uses a numerical function also called yoke-structures (written in C and run remotely on a Vax) to prune the legal locations for both anchorees to include only locations that satisfy constraints between them. It modifies their locations attributes accordingly.

(next page)

Figure 10
(caption on
preceding page)

Name: Yoke-Structures

Trigger Conditions:
```
((\$EVENT-LEVEL-IS STRUCTURAL.SOLID)
 ($EVENT-TYPE-IS Modify)
 ($CHANGED-ATTRIBUTE-IS APPLIED-CONSTRAINTS)
 ($SET Possible-Combinations (Get-Possible-Combinations $TRIGGER-OBJECT)))
```

Context Variables:
```
((PS-Anchor Anchoree1 Anchoree2) Possible-Combinations)
```

Preconditions:
```
(($SET Yoking-Info (There-Is-Yoking-Info-For Anchoree1 Anchoree2))
 ($VALUE Anchoree1 'Applied-Constraints)
 ($VALUE Anchoree2 'Applied-Constraints))
```

Obviation Conditions: NIL

KS Variables:
```
((NewLocLabelForAnchoree1 (Generate-LocTableLabel PS-Anchor Anchoree1
    (LENGTH ($VALUE Anchoree1 'Legal-Orientations))))
 (NewLocLabelForAnchoree2 (Generate-LocTableLabel PS-Anchor Anchoree2
    (LENGTH ($VALUE Anchoree2 'Legal-Orientations))))
 (Descriptor1 (Make-Descriptor-For-Yoke PS-Anchor Anchoree1 Anchoree2))
 (Descriptor2 (Make-Descriptor-For-Yoke PS-Anchor Anchoree2 Anchoree1)))
```

Actions:
```
((1 (T)
    (EXECUTE ($SET YokeResult (Yoke-Structures PS-Anchor
          Anchoree1 Anchoree2
          (CADAR (LAST ($VALUE Anchoree1 'Legal-Orientations)))
          (CADAR (LAST ($VALUE Anchoree2 'Legal-Orientations)))
          NewLocLabelForAnchoree1 Descriptor1
          NewLocLabelForAnchoree2 Descriptor2
          (LENGTH Yoking-Info) Yoking-Info VanderWaalsCheck?))))
 (2 (T)
    (PROPOSE changetype MODIFY object Anchoree1 attributes
          ((Legal-Orientations (APPEND
                  ($VALUE Anchoree1 'Legal-Orientations)
                  (LIST (LIST 'Yoke NewLocLabelForAnchoree1
                  (Car YokeResult) Descriptor1)))
          (Applied-Constraints (APPEND
                  ($VALUE Anchoree1 'Applied-Constraints)
                  (LIST Yoking-Info))))))
 (3 (T)
    (PROPOSE changetype MODIFY object Anchoree2 attributes
          ((Legal-Orientations (APPEND
                  ($VALUE Anchoree2 'Legal-Orientations)
                  (LIST (LIST 'Yoke NewLocLabelForAnchoree2
                  (Car YokeResult) Descriptor2)))
          (Applied-Constraints (APPEND
                  ($VALUE Anchoree2 'Applied-Constraints)
                  (LIST Yoking-Info))))))
```

Name - KSAR50
Trigger-Event - ANCHOR-HELIX modifying attributes of HELIX1
ContextVars - ((PS-Anchor Helix1)
(Anchoree1 Helix3)
(Anchoree2 Helix2))
KS - Yoke-Structures
BoundVars - ((NewLocLabelForAnchoree1 Hel1inHel3-5)
(NewLocLabelForAnchoree2 Hel1inHel2-4)
(Descriptor1 Yoke-Helix3-and-Helix2-around-Helix1)
(Descriptor2 Yoke-Helix2-and-Helix3-around-Helix1))
ExecutableCycle - 18
ScheduledCycle - NIL
ExecutedCycle - NIL
Status - EXECUTABLE

Figure 11. A Yoke-Structures KSAR. Yoke-Structures has been triggered by a modification of helix1's applied-constraints. This KSAR represents the blackboard context in which helices 2 and 3 have constraints with one another and with helix1. Since both helices have previously identified locations, the KSAR is executable.

EventName - EVENT64
ObjectName - Helix3
EventCycle - 27
ChangeType - MODIFY
EventLevel - SOLUTION.SOLID
Changes - ((Locations (. . .))
(Applied-Constraints (. . .)))
Creator - KSAR50

Figure 12. A Yoke-Structures Event. On BB1 cycle 27, KSAR50 was executed and Yoke-Structures modified helix3's locations and applied-constraints.

```
Strategy
    Develop-PS-of-Best-Anchor
       |------------------------------------------------------------------->

Focus
    Create-Best-Anchor-Space
       |-------------------|
    Position-All-Structures-Helix1
                           |----------------------------------------------->

Heuristic
    Prefer-Rigid-Anchors
        |------------------|
    Prefer-Long-Anchors
         |-----------------|
    Prefer-Constraining-Anchors
          |----------------|
    Prefer-PS-Anchor-Is-Helix1
                              |-------------------------------------------->
    Prefer-Rigid-Anchorees
                               |------------------------------------------->
    Prefer-Long-Anchorees
                                |------------------------------------------>
    Prefer-Constraining-Anchorees
                                 |----------------------------------------->
    Prefer-Constrained-Anchorees
                                  |---------------------------------------->
    Prefer-Strong-Constraints
                                   |--------------------------------------->

Cycle |----------|-----------|-----------|------------|------------|------------|
      0          5           10          15           20           25           30
```

Figure 13. A PROTEAN Control Plan. In attempting to solve the lac-repressor headpiece, PROTEAN decides to pursue a strategy in which it develops a single partial solution around the best available anchor. It immediately begins to implement this strategy, focusing on actions that create the best anchor space. In particular, it favors actions that give preference to long, rigid, constraining structures as the anchor. As indicated by the directed lines associated with this focus, PROTEAN performs these kinds of actions for approximately six BB1 cycles, until it identifies helix1 as the best anchor. PROTEAN then implements the next phase of its strategy, focusing on actions that position other structures relative to helix1. In particular, it favors actions that position long, rigid, constraining, constrained anchorees with strong constraints. As indicated by the directed lines associated with this focus, PROTEAN performs these kinds of actions until at least BB1 cycle 30.

```
Protein-Name - Lac-Repressor-Headpiece
Primary-Structure - (MET1 LYS2 PRO3 VAL4 THR5 LEU6 TYR7 ASP8 VAL9 ALA10
                     GLU11 TYR12 ALA13 GLY14 VAL15 SER16 TYR17 GLN18 THR19
                     VAL20 SER21 ARG22 VAL23 VAL24 ASN25 GLN26 ALA27 SER28
                     HIS29 VAL30 SER31 ALA32 LYS33 THR34 ARG35 GLU36 LYS37
                     VAL38 GLU39 ALA40 ALA41 MET42 ALA43 GLU44 LEU45 ASN46
                     TYR47 ILE48 PRO49 ASN50 ARG51)
Secondary-Structure - ((Coil  1 MET1  THR5)
                     (Helix 1 LEU6  GLY14)
                     (Coil  2 VAL15 SER16)
                     (Helix 2 TYR17 ASN25)
                     (Coil  3 GLN26 ARG35)
                     (Helix 3 GLU36 LEU45)
                     (Coil  4 ASN46 ARG51))
NOEs - ((1  VAL4  3 TYR17 5)
        (2  VAL4  3 TYR47 5)
        (3  LEU6  4 TYR17 5)
        (4  LEU6  4 MET42 5)
        (5  LEU6  4 TYR47 5)
        (6  VAL9  3 TYR47 5)
        (7  ALA10 2 TYR17 5)
        (8  TYR12 5 ALA32 2)
        (9  TYR12 5 ALA40 2)
        (10 TYR12 5 ALA41 2)
        (11 TYR12 5 MET42 5)
        (12 TYR12 5 LEU45 4)
        (13 VAL15 3 TYR47 5)
        (14 TYR17 5 MET42 5)
        (15 VAL24 3 TYR47 5)
        (16 VAL30 3 MET42 5)
        (17 MET42 5 TYR47 5))
```

Figure 14. PROTEAN's Representation of the Lac-Repressor Headpiece Problem. The attribute, primary-structure, lists the lac-repressor's defining sequence of amino acids. Secondary-structure lists each random coil and alpha helix in the protein, along with first and last amino acids in the corresponding subsequence of the primary structure. NOEs lists all observed NOE measurements, along with the associated pairs of amino acids and atoms.

Title - DEVELOP-PS-OF-BEST-ANCHOR
Description - "Position solid structures relative to the single best anchor."
Rationale - "For small proteins, a single partial solution will do."
Weight - 10
Procedure-Type - Follow-Sequence
Procedure-Data - (CREATE-BEST-ANCHOR-SPACE POSITION-ALL-STRUCTURES)
Expired-Prescription - NIL
Current-Prescription - (CREATE-BEST-ANCHOR-SPACE)
Goal - (ALL-PRESCRIPTIONS-FOLLOWED)
Status - Operative
First-Cycle - 1
Last-Cycle - NIL

Figure 15. An Illustrative PROTEAN Strategy: Develop-PS-of-Best-Anchor. The attribute, procedure-type, indicates that this strategy should be refined with a generic BB1 procedure called follow-sequence, which successively establishes each element of the strategy's procedure-data as the strategy's current-prescription. The goal indicates that the strategy should remain operative until PROTEAN has established all elements of the procedure-data as the current-prescription and achieved each of their individual goals.

Name: Initialize-Prescription

Trigger Conditions:

```
((\$EVENT-LEVEL-IS CONTROL-PLAN.STRATEGY)
 (\$EVENT-TYPE-IS ADD)
 (EQ (\$VALUE \$TRIGGER-OBJECT 'Procedure-Type) 'Follow-Sequence))
```

Context Variables: NIL

Preconditions: (T)

Obviation Conditions: NIL

KS Variables: NIL

Actions:

```
((1 (T)
    (PROPOSE changetype MODIFY object $TRIGGER-OBJECT attributes
         ((Current-Prescription (CAR
                 ($VALUE $TRIGGER-OBJECT 'Procedure-Data)))))))
```

Figure 16. A Generic Control Knowledge Source. Initialize-Prescription is triggered by the appearance of a new strategy whose procedure-type is follow-sequence. When executed, it establishes the first element of the strategy's procedure-data as its current-prescription.

```
Title - Create-Best-Anchor-Space
Description - "Create an anchor space for the most constraining solid-anchor."
Rationale - "The most constraining solid-anchor will restrict the search space for
                positioning other structures."
Weight - 1
Heuristics - (PREFER-RIGID-ANCHORS   PREFER-LONG-ANCHORS
                PREFER-CONSTRAINING-ANCHORS)
Goal - ($FIND (QUOTE STRUCTURAL.SOLID-ANCHOR)
                (QUOTE ((ROLE (QUOTE ANCHOR))
                        (ACTIVATED? T))))
Select-Update - ($FIND (QUOTE STRUCTURAL.SOLID-ANCHOR)
                        (QUOTE ((ROLE (QUOTE ANCHOR))
                                (ACTIVATED? T))))
Focus-Status - Operative
Stability - Dynamic
First-Cycle - 3
Last-Cycle - NIL
```

Figure 17. PROTEAN's First Focus. Create-Best-Anchor-Space identifies three heuristics for choosing an anchor: Prefer-Long-Anchors, Prefer-Constraining-Anchors, and Prefer-Rigid-Anchors (see Figure 18). Its goal specifies that the scheduler should use these heuristics to choose pending KSARs until PROTEAN has established an anchor. Its select-update specifies that, when this goal is achieved, the established anchor should be passed back to the superordinate strategy.

Title - Prefer-Rigid-Anchors
Description - "Prefer rigid structures as anchors."
Rationale - "They can be positioned more precisely and, therefore, they impose stronger constraints on the positioning of anchorees."
Weight - 8
Function -
```
(if (EQ ($VALUE KSAR 'ObjectName) 'Helix)
        then 100
    elseif (EQ ($VALUE KSAR 'ObjectName) 'Beta-Sheet)
        then 50 else 0)
```
Heuristic-Status - Operative
First-Cycle - 7
Last-Cycle - NIL
Stability - Stable

Figure 18. A PROTEAN Heuristic. Prefer-Rigid-Anchors specifies that KSARs should get ratings of 100, 50, or 0, depending upon whether their actions operate on anchors that are alpha helices, beta sheets, or random coils. The weight, 8, indicates that this heuristic is rather important, on a scale 1-10.

Name - KSAR15
Trigger-Event - POST-SOLID-ANCHORS adding Helix1
ContextVars - NIL
KS - Activate-Anchor-Space
BoundVars - ((PS-Anchor Helix1))
ExecutableCycle - 7
ExecutedCycle - NIL
ScheduledCycle - 16
Status - EXECUTABLE
Focus - FOCUS1: Create-Best-Anchor-Space
Ratings - Prefer-Rigid-Anchors 100
Prefer-Long-Anchors 60
Prefer-Constraining-Anchors 33.33
Prefer-Activate-Anchor-Space 100
Weighted-Total - 2564

Figure 19. Ratings for KSAR15 against PROTEAN's First Focus. KSAR15 represents the knowledge source Activate-Anchor-Space triggered by the creation of an object called helix1. Its attribute, weighted-total, shows its overall rating against PROTEAN's first focus decision (see Figure 17). This value combines KSAR15's ratings against each of the focus decision's component heuristics. For example, since KSAR15 specifies helix1 as the anchor, it gets a rating of 100 against the heuristic called Prefer-Rigid-Anchors (see Figure 18). This rating contributes to the weighted-total in proportion to its weight, 8.

Title - POSITION-ALL-STRUCTURES-HELIX1
Description - "Opportunistically apply all available constraints to anchor and yoke all solid structures in the context of the strategically selected anchor."
Rationale - "Opportunistic selection of anchorees, constraints, and operations (anchoring versus yoking) yields the most restricting individual actions."
Weight - 1
Heuristics - (PREFER-STRATEGICALLY-SELECTED-ANCHOR
PREFER-RIGID-ANCHOREES
PREFER-LONG-ANCHOREES
PREFER-CONSTRAINED-ANCHOREES
PREFER-CONSTRAINING-ANCHOREES
PREFER-STRONG-CONSTRAINTS)
Goal - (ALL-STRUCTURES-POSITIONED-FOR 'HELIX1)
Select-Update - NIL
Focus-Status - Operative
Stability - Dynamic
First-Cycle - 11
Last-Cycle - NIL

Figure 20. PROTEAN's Second Focus. Position-All-Structures specifies six heuristics for positioning structures relative to the chosen anchor, helix1: Prefer-Rigid-Anchorees, Prefer-Long-Anchorees, Prefer-Constraining-Anchorees, Prefer-Constrained-Anchorees, Prefer-Strong-Constraints, and Prefer-PS-Anchor-is-Helix1. Its goal specifies that the scheduler should use these heuristics to choose pending KSARs until PROTEAN has positioned all structures relative to helix1.

Name - KSAR34
Trigger-Event - ACTIVATE-ANCHORSPACE modifying attributes of HELIX1
ContextVars - NIL
KS - Add-Anchoree-to-Anchorspace
BoundVars - ((PS-Anchor Helix1)
(PS-Anchoree Helix2)
. . .)
ExecutableCycle - 16
ExecutedCycle - NIL
ScheduledCycle - NIL
Status - EXECUTABLE
Focus - FOCUS2: Position-All-Structures-Helix1
Ratings - Prefer-PS-Anchor-Is-Helix1 100
Prefer-Rigid-Anchorees 100
Prefer-Long-Anchorees 70
Prefer-Constraining-Anchors 20
Prefer-Constrained-Anchorees 100
Prefer-Strong-Constraints 0
Weighted Total - 2750

Figure 21. Ratings for KSAR34 against PROTEAN's Second Focus. KSAR34 represents the knowledge source Add-Anchoree-to-Anchor-Space triggered by the establishment of helix1 as the anchor, with helix2 as a potential anchoree. Its weighted-total rating against PROTEAN's second focus decision (see Figure 20) combines its ratings against each of the focus decision's component heuristics. For example, since KSAR34 specifies helix2 as the anchoree, it gets a rating of 100 against the Heuristic called Prefer-Rigid-Anchorees.

I recommend KSAR15: Activate-Anchor-Space modifying attributes of Helix1

> **Why**

I am recommending KSAR15 because I am attempting to

H2 (100) Prefer rigid structures as anchors
H3 (60) Prefer Anchors that represent longer segments of the primary sequence
H4 (33.33) Prefer Anchors that have many constraints with many other structures

> **Explain H2**

I am attempting to prefer rigid structures as anchors because they can be positioned more precisely and, therefore, they impose stronger constraints on the positioning of anchorees.

> **Why**

I am using H2, H3, and H4 because I am attempting to

F1 Create an anchor space for the most constraining solid-anchor.

> **Explain**

I am attempting to create an anchor space for the most constraining solid-anchor because the most constraining solid-anchor will restrict the search space for positioning other structures.

> **Why**

I am using F1 because I am attempting to

S1 Position solid structures relative to the single best anchor.

> **Explain**

I am attempting to position solid structures relative to the single best anchor because for small proteins, a single partial solution will do.

Figure 22. Explanation of the Decision to Perform KSAR15.

```
I recommend:
    KSAR56: Yoke-Structures

> Override with KSAR55.

I am performing:
    KSAR55: Anchor-Helix

. . .

I would like to understand why you prefer KSAR55 over KSAR56:

Is it because you prefer:
    (a) KS = Anchor-Helix over KS = Yoke-Structures
    . . .

> a

I am modifying my current plan to include a new heuristic:

  (if (EQ ($VALUE KSAR 'KS) 'Anchor-Helix) then 100 else 0

Is this OK?

> Yes

What should I call the new heurstic?

> Prefer-Anchoring-Over-Yoking

Shall I modify the knowledge source: Build-PAs?

> Yes

I am modifying the knowledge source: Build-PAs.
```

Figure 23 MARCK Learns a New Control Heuristic.

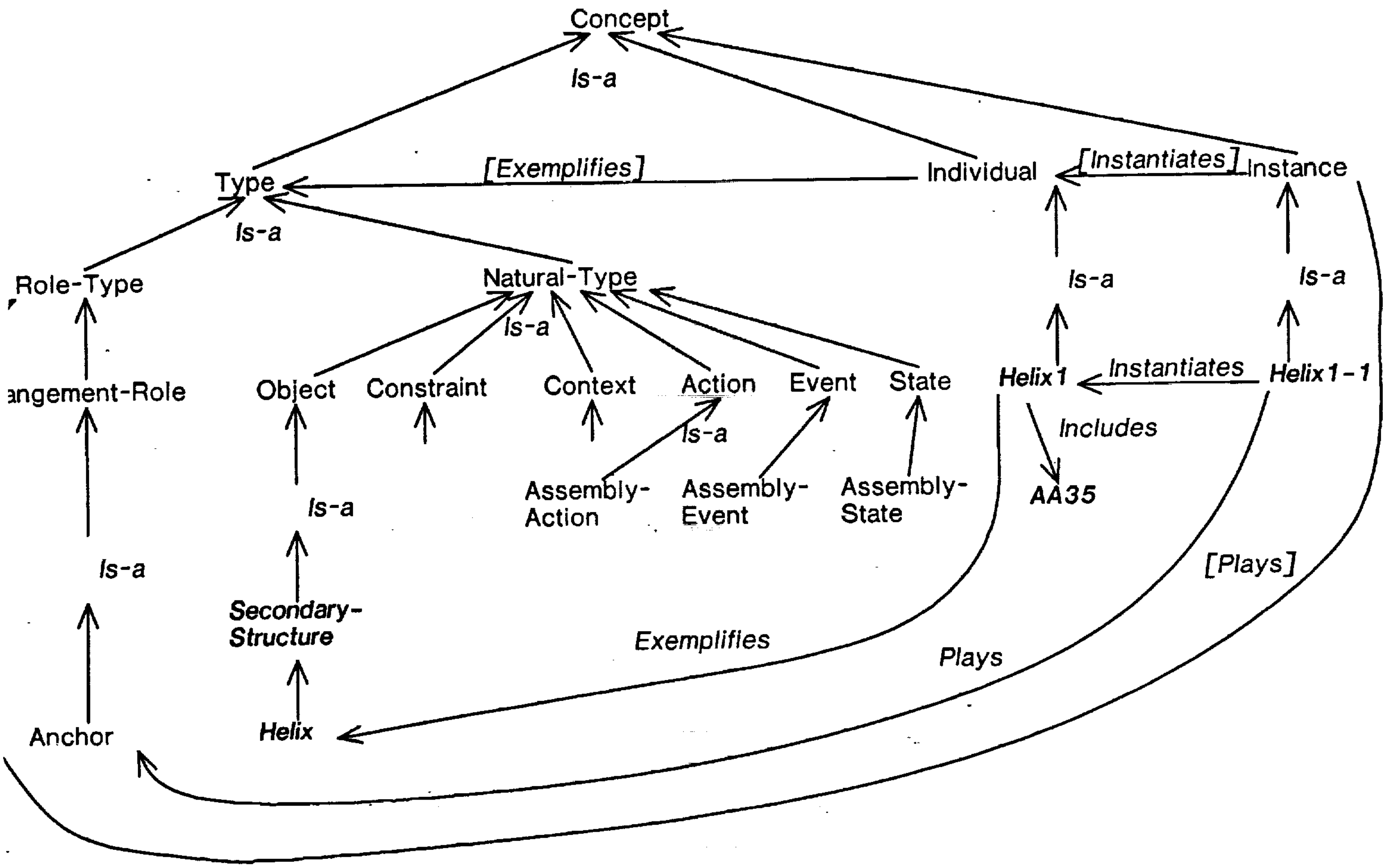


Figure 24. Schematic Overview of ACCORD's Conceptual Network. Concept types (e.g., helix, secondary-structure) intensionally define generic concepts by means of is-a links. Concept individuals (e.g., helix1) exemplify particular concept types. Concept instances (e.g., helix1-1) instantiate particular individuals to play particular roles (e.g., anchor) in particular contexts. Bracketed links indicate legal links (e.g., a concept individual may exemplify a concept type), while unbracketed links indicate actual links (e.g., the individual helix1 actually does examplify the type helix). Concepts can have attributes whose values are static or procedural. Both attributes and link relations are inheritable. PROTEAN-specific concepts in this figure appear in bold type.

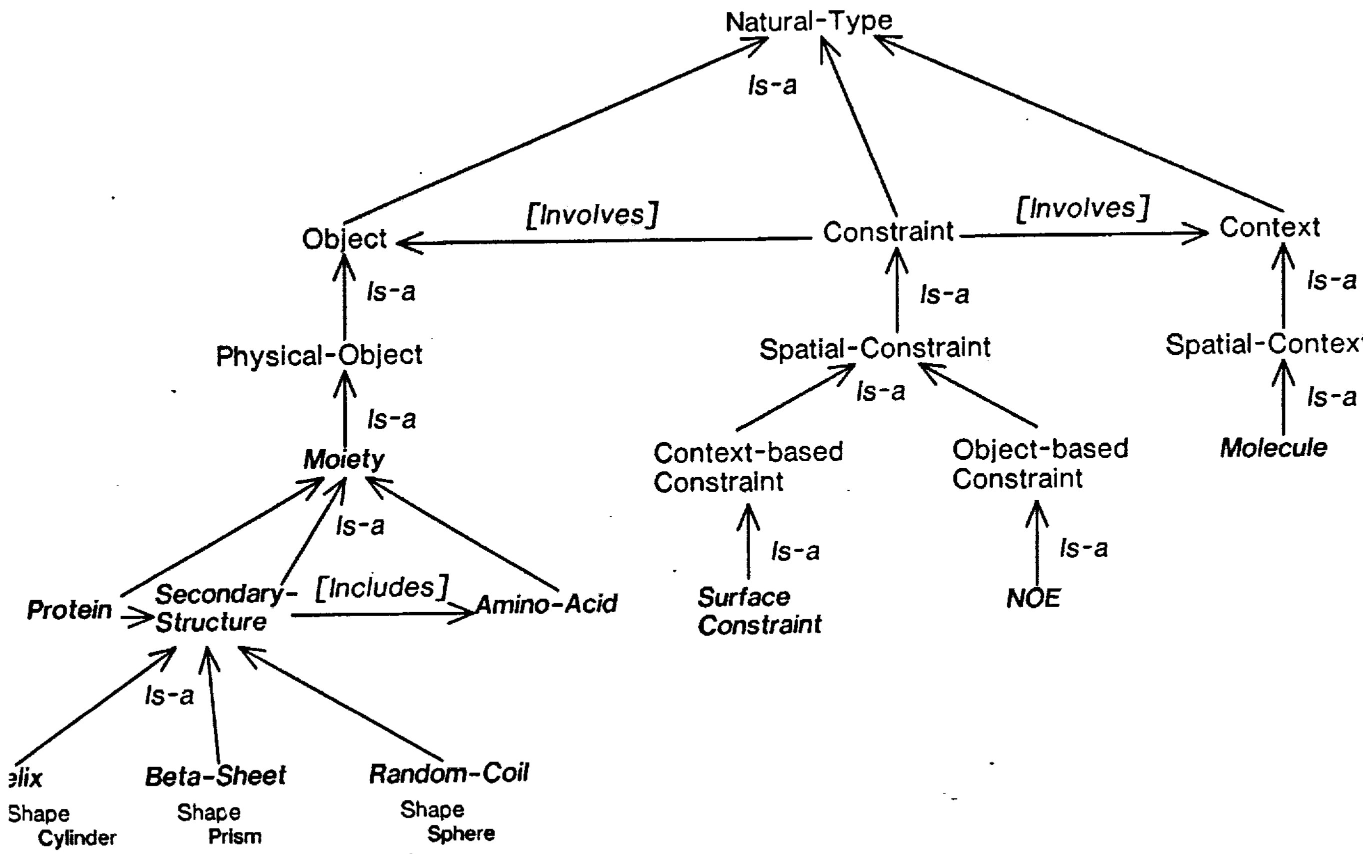


Figure 25. ACCORD's Skeletal Branches for Objects, Contexts, and Constraints. ACCORD requires specification of the objects, contexts, and constraints that figure in arrangement problems in particular domains. In general, particular constraints can involve particular objects or contexts. PROTEAN-specific entities appear in bold type.

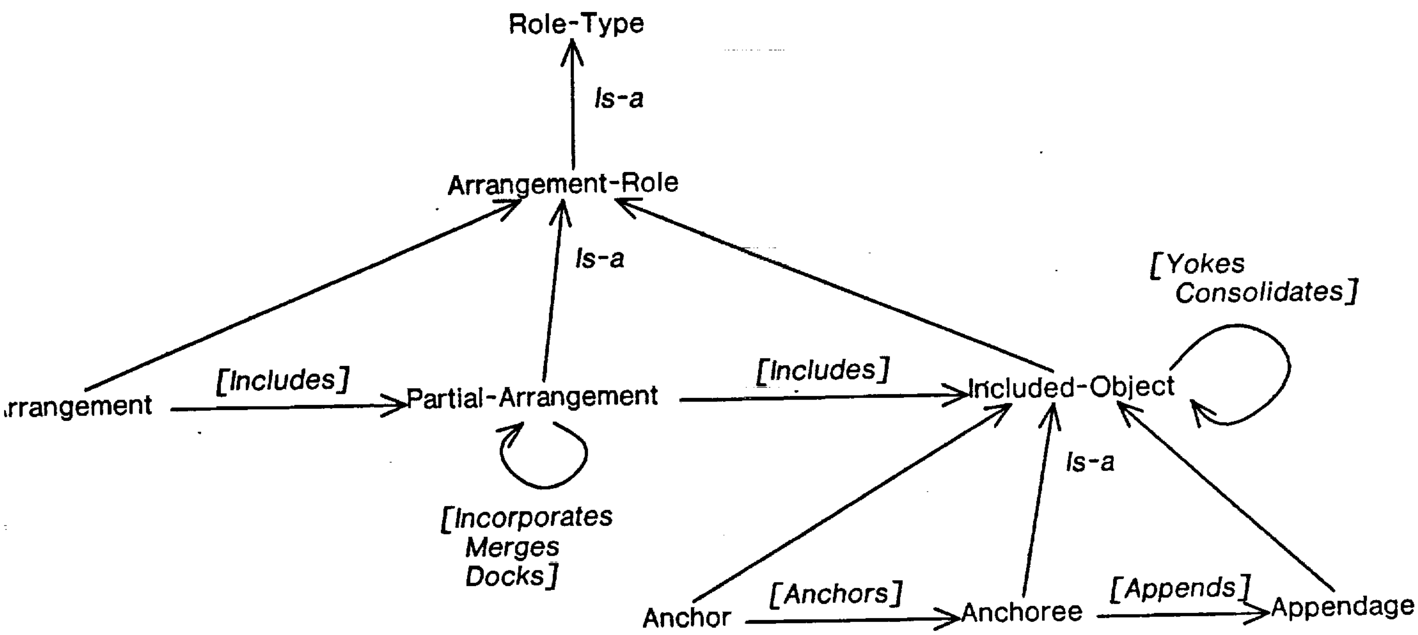


Figure 26. ACCORD's Arrangement-Role Types. An arrangement is a complete solution to an arrangement problem and may include one or more partial arrangements. A partial-arrangement is a partial solution that includes a subset of the objects, constraints, and contextual regions specified in the problem. Particular partial-arrangements can incorporate, merge, or dock with one another. Included-objects can serve as anchors, anchorees, or appendages within a partial-arrangement. An anchor can anchor an anchoree. An anchoree can append an appendages. In addition, included-objects can yoke or consolidate with one another.

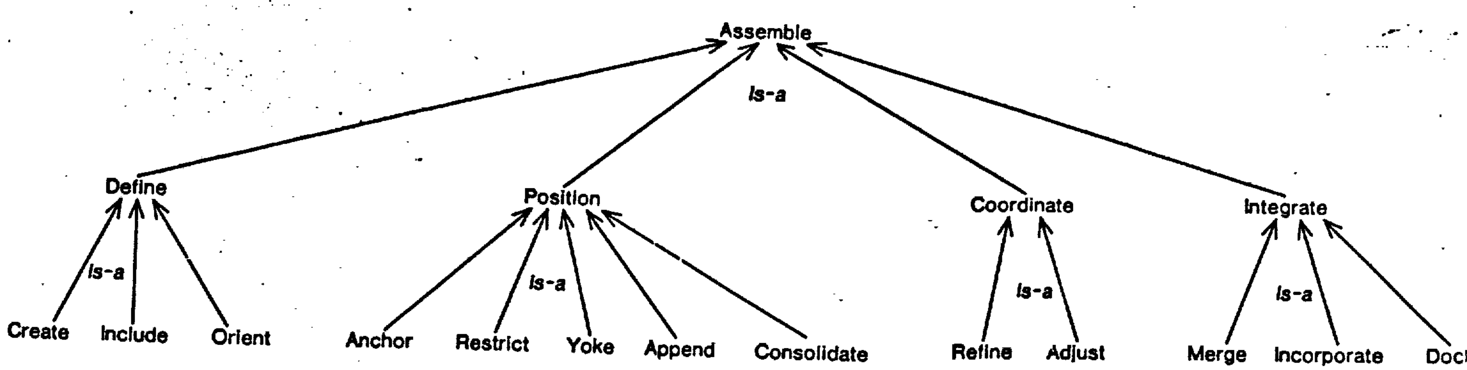


Figure 27. ACCORD's Type Hierarchy of Arrangement-Assembly Root Verbs. Assemble has four subtypes. Defining a partial arrangement involves creating a partial arrangement, including objects in it, and orienting the partial arrangement about a selected anchor. Positioning objects within a partial arrangement may involve anchoring, restricting, yoking, appending, or consolidating them. Coordinating partial arrangements may involve refining them at lower levels of abstraction or adjusting them at higher levels of abstraction. Integrating partial arrangements may involve merging those that have a common anchor, incorporating one partial arrangement into another one that shares a common object, or docking those that include objects that constrain one another.

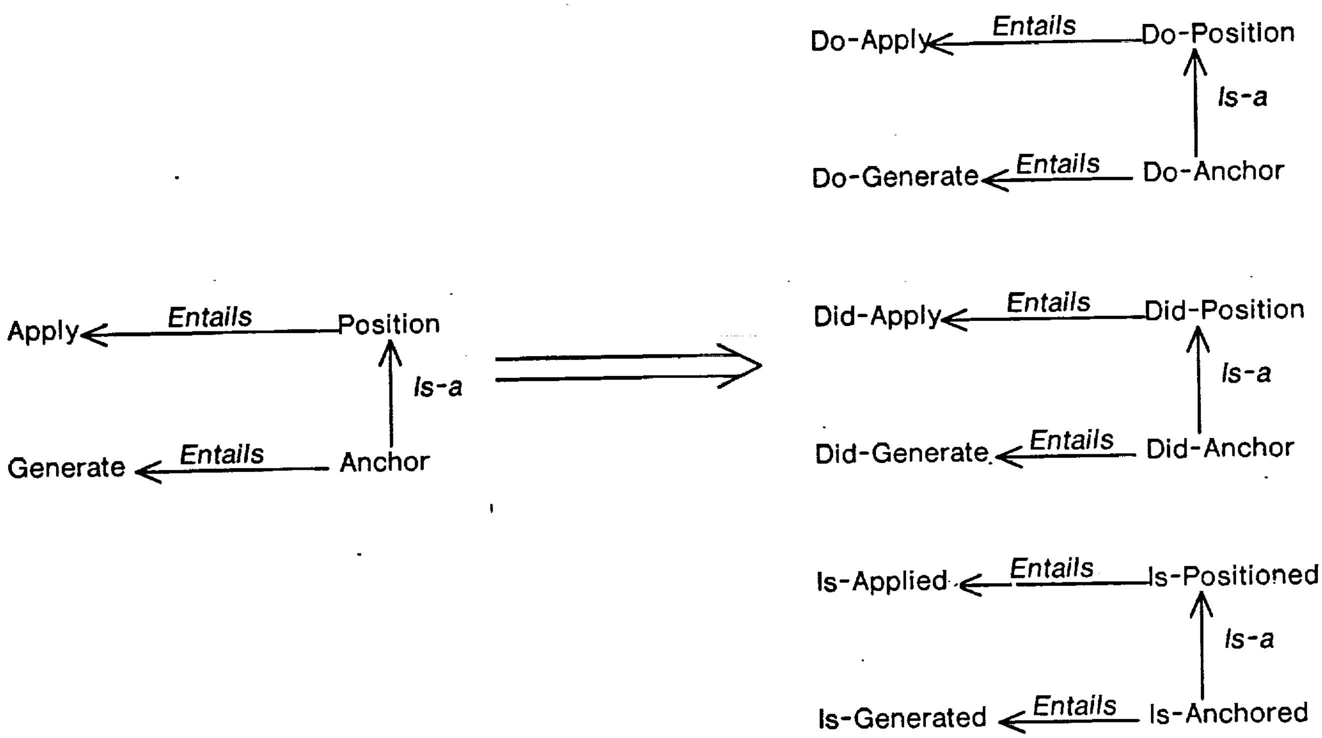


Figure 28. Homologous Action, Event, and State Subnetworks. The root verb hierarchy underlies homologous action, event, and state type hierarchies, distinguished by verb tense. Do-<verb> signifies an action. Did-<verb> signifies an event. Is-<verbed> signifies a state. Implicit $<links> indicate, for example, that do-anchor actions $entail do-apply actions.

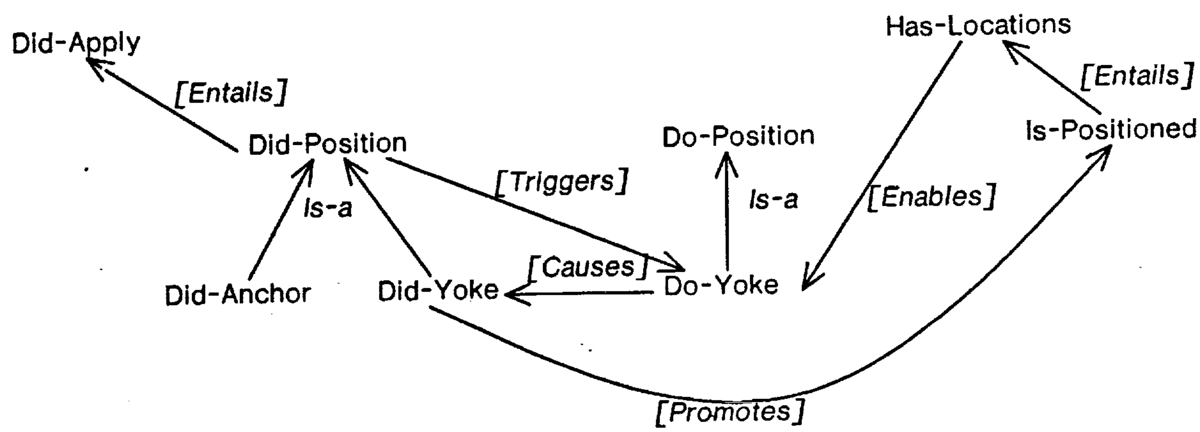


Figure 29. Some Legal Relations among Actions, Events, and States. Did-position events trigger do-yoke actions, which must be enabled by has-locations states. When executed, do-yoke actions cause did-yoke events, which promote is-positioned states. Implicit $<link> relations indicate, for example, that did-anchor events trigger do-yoke actions and that do-yoke actions cause did-apply events.

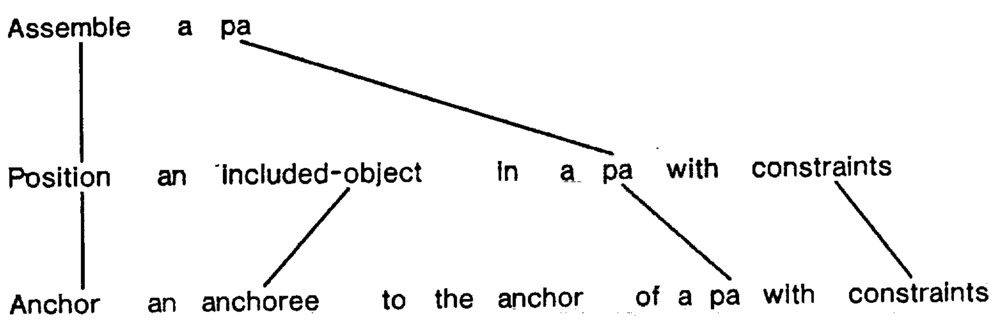


Figure 30. Partial Matches between Assemble, Position, and Anchor Templates. Partial matches identify semantically corresponding formal parameters in all pairs of templates. In theses examples: Assemble, position, and anchor all represent verb keywords. Included-object and anchoree represent objects being positioned. All parameters called pa refer to the partial arrangement. Parameters called constraints represent constraints to be applied.

ACCORD Template: Anchor Anchoree to Anchor in PA with Constraints.

BB1 Template:

```
((1 (T)
        ((EXECUTE ($Set Constraints (CONSTRAINTS-IN Constraints)))
         (EXECUTE ($Set CSS-Anchor-Results (CDR (CSS-ANCHOR Anchoree
                Anchor PA Constraints))))
         (PROPOSE changetype MODIFY object Anchoree attributes
                CSS-ANCHOR-RESULTS))))
```

PROTEAN CSS-ANCHOR Function:

```
(PROG (AbTable PObject PAnchor PConstraints Sample-Vector Description
        CalcLocAns DescribeAns)
    (SETQ AbTable (CSS-GENERATE-TABLE-NAME Object Anchor
        Constraints PA 'Anchor)
    (SETQ PObject ($SHORT-NAME ($OBJECT Object 'Instantiates)))
    (SETQ PAnchor ($SHORT-NAME ($OBJECT Anchor 'Instantiates)))
    (SETQ PConstraints ($SHORT-NAME Constraints))
    (SETQ Sample-Vector '(2 2 2 30 30 30))
    (SETQ Description (LIST 'Anchor PObject 'to PAnchor))
    (SETQ CalcLocAns (GS-CALCULATE-LOCATIONS AbTable NIL PAnchor
        PObject PConstraints NIL Description Sample-Vector NIL))
    (IF (NULL (CAR CalcLocAns))
        THEN (RETURN CalcLocAns))
    (SETQ DescribeAns (GS-DESCRIBE-LOCATIONS AbTable PAnchor
        PObject PConstraints 'GS-CALCULTATE-LOCATIONS
        (DATE) Description))
    (RETURN (CDR DescribeAns)))
```

Figure 31. ACCORD and BB1 Templates for the Do-Anchor Action. Both templates refer to the same parameters, which can be instantiated to define specific action patterns. The ACCORD template is essentially a macro for the more complex underlying BB1 program of rules. Note that all application-specific routines for constraint satisfaction are inserted indirectly through calls to ACCORD's generic CSS-<extension> functions.

ACCORD Template: Is-Anchored Anchoree to Anchor in PA with Constraints.

BB1 Template:

```
((EQ ($OBJECT Anchoree 'Anchored-by) Anchor)
 (FMEMB Anchor ($OBJECTS PA 'Includes))
 ($OBJECT Anchoree 'Located-by)
 (EQ ($VALUE ($OBJECT Anchoree 'Located-by) 'Constraint-Set-Used)
    Constraints))
```

Figure 32. ACCORD and BB1 Templates for the Is-Anchored State. Both templates refer to the same parameters, which can be instantiated to define specific state patterns. The ACCORD template is essentially a macro for the more complex underlying BB1 program of access functions.

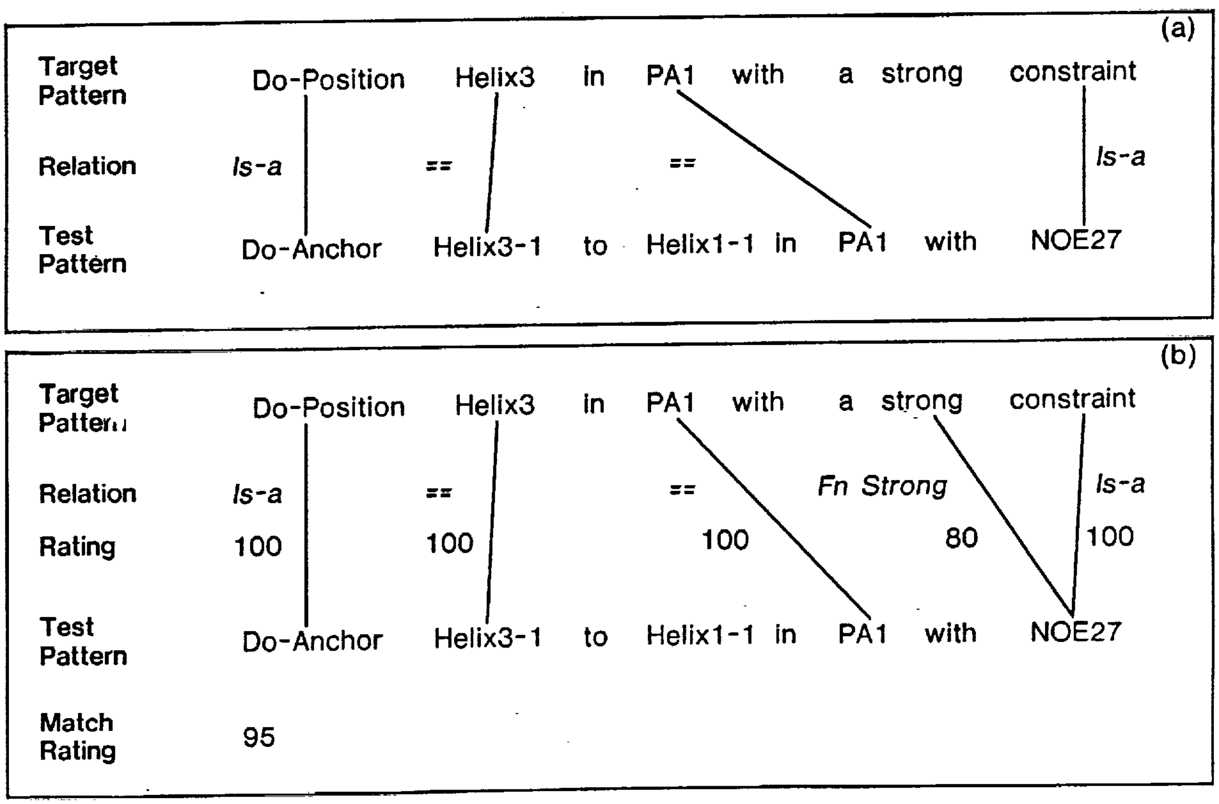


Figure 33. Matching Two Action Patterns. (a) The test pattern produces a perfect match to the target pattern because: Do-anchor is-a do-position action. Helix3-1 is helix3-1. Pal is pal. NOE1 is-a constraint. (b) The match rating, 95, combines component ratings for each parameter and modifier in the target pattern, proportionate to their weights. In this case, the perfect match entails ratings of 100 for each parameter and NOE27 rates 80 against the modifer, strong.

```
Generate  X such that:
          Is-a X Long Helix
          Plays X Included-Objects
          Is-Positioned X

Is-a X (Long) Helix
          -> ($ALL-OBJECTS Helix 'Can-be-a)
              = (Helix1 Helix2 Helix3
                  Helix1-1 Helix2-1 Helix3-1)

Plays X Included-Object
          -> (Helix1-1 Helix2-1 Helix3-1)

Is-Positioned X
          -> (Helix1-1 Helix2-1 Helix3-1)

Is-a X Long Helix
          -> ((Helix1-1 (90)) (Helix3-1 (70)) (Helix2-1 (40)))
```

Figure 34. Generation of Parameter Values. This set of expressions generates all long helixes that are positioned in some partial arrangement, best first. First, the generator generates all legal values for X to instantiate the state, Is-a helix. Then it prunes this set to include only legal values of X to instantiate the state, Plays X included-object. Then it prunes the reduced set to include only legal values of X to instantiate the state, Is-positioned X. Finally, it orders the remaining set according to the rating of each value in the phrase, Long X.

ACCORD Template: Anchor Anchoree to Anchor in PA with Constraints.

BB1 Template:

```
((1 (T)
        ((EXECUTE ($Set Constraints (CONSTRAINTS-IN Constraints)))
         (EXECUTE ($Set CSS-Anchor-Results (CDR (CSS-ANCHOR Anchoree
                Anchor PA Constraints))))
         (PROPOSE changetype MODIFY object Anchoree attributes
                CSS-ANCHOR-RESULTS))))

                [CSS-ANCHOR . . . ]
```

ACCORD Pattern: Do-Anchor Helix2-1 to Helix1-1 in PA1 with CSet1.

BB1 Pattern:

```
((1 (T)
        ((EXECUTE ($Set Constraints (CONSTRAINTS-IN CSet1)))
         (EXECUTE ($Set CSS-Anchor-Results (CDR (CSS-ANCHOR Helix2-1
                Helix1-1 PA1 CSet1))))))
         (PROPOSE changetype MODIFY object Helix2-1 attributes
                CSS-ANCHOR-RESULTS)

                [CSS-ANCHOR . . . ]
```

Figure 35. Translation of Action Patterns. The translator substitutes the parameter values in the ACCORD pattern for the corresponding parameters in the BB1 template.

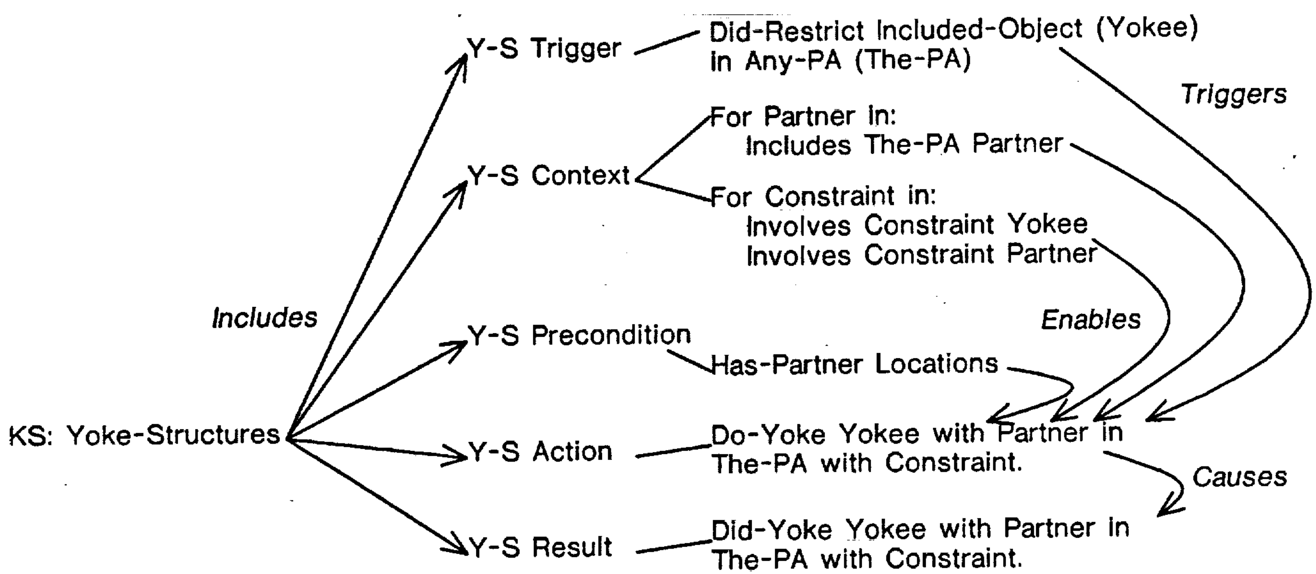


Figure 36. A Domain Knowledge Source in ACCORD. Yoke-Structures (see Figure 10) is represented as a conceptual network comprising each knowledge source component (e.g., trigger precondition), each component's constituent action, event, and state patterns, and appropriate links among them. Each of these patterns exemplifies a particular action, event, or state type. For example, Yoke-Structures's trigger event, *Did-restrict included-object (yokee) in any-pa (the-pa)* exemplifies the did-restrict event. For simplicity, these exemplifies links do not appear in the illustration.

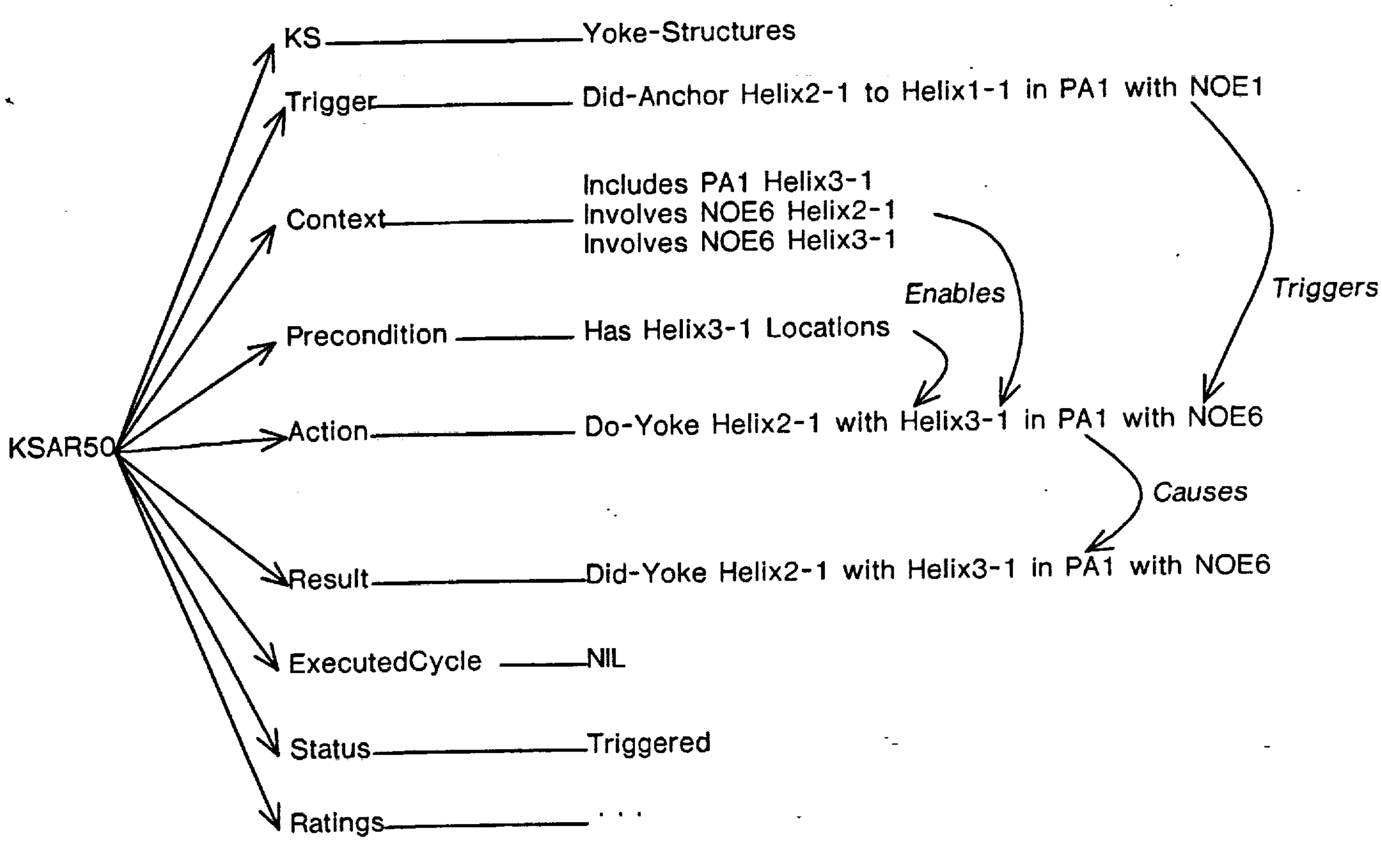


Figure 37. A KSAR in ACCORD. This Yoke-Structures KSAR (see Figure 11) is represented as a conceptual network comprising each knowledge source component, each component's action, event, and state patterns, and appropriate links among them. Each of these patterns matches or instantiates the corresponding pattern in the Yoke-Structures knowledge source. For example, KSAR50's trigger event, *Did-anchor helix2-1 to helix1-1 in pa1 with NOE1* matches Yoke-Structures's trigger event, *Did-restrict included-object (yokee) in any-pa (the-pa)* because did-anchor entails did-restrict, helix2-1 plays included-object and pa1 plays pa. Similarly, KSAR50's action, *Do-yoke helix2-1 with helix3-1 in pa1 with NOE6* instantiates Yoke-Structures's action, *Do-yoke yokee with partner in the-pa with constraint* because helix2-1 is the bound value of yokee, helix3-1 is the bound value of partner, pa1 is the bound value of the-pa, and NOE6 is the bound value of constraint. Again, for simplicity, these links do not appear in the illustration.

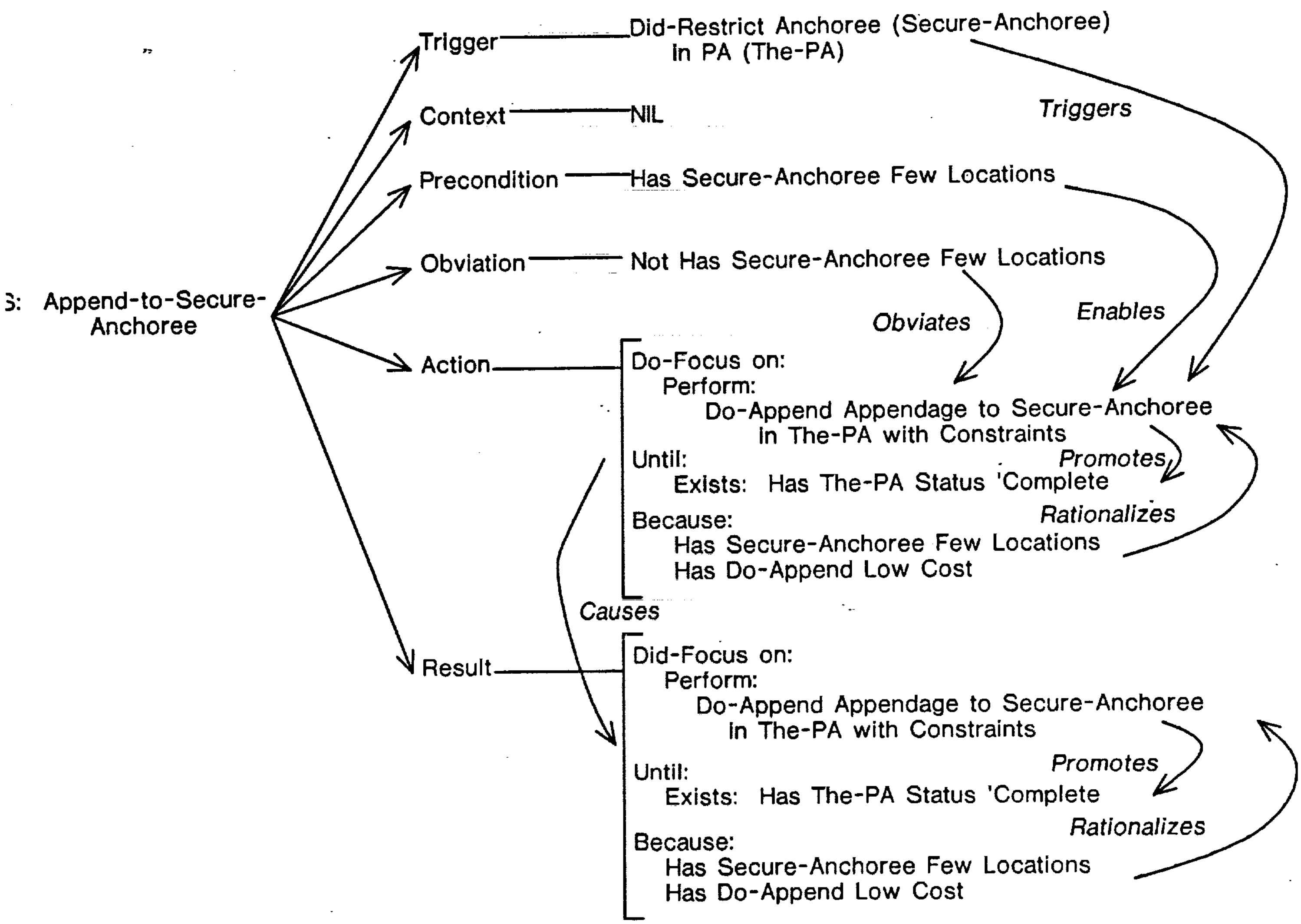
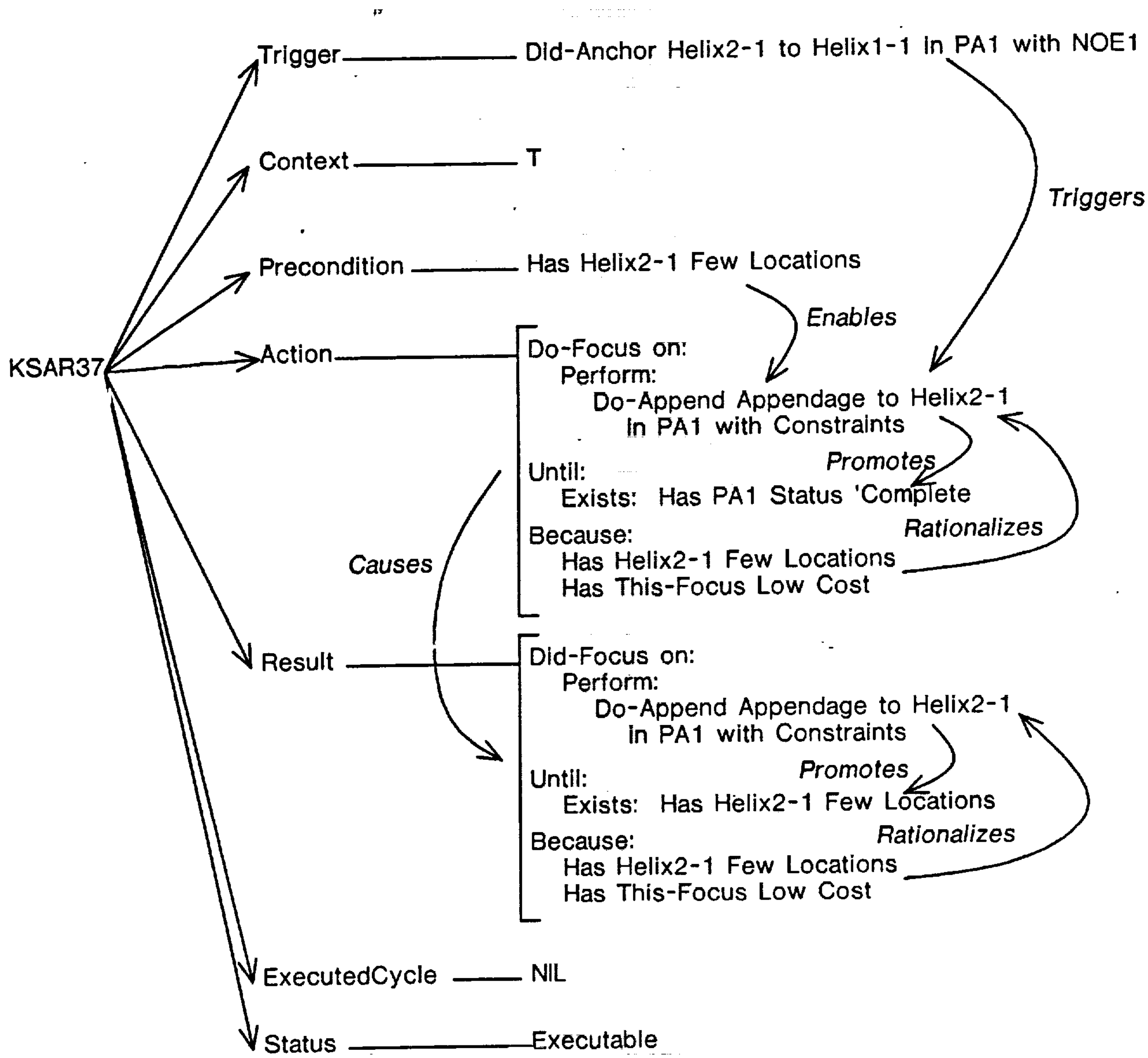


Figure 38. A Control Knowledge Source in ACCORD. Append-to-Secure-Anchoree is represented as a conceptual network comprising each knowledge source component (e.g., trigger precondition), each component's constituent action, event, and state patterns, and appropriate links among them. Each of these patterns exemplifies a particular action, event, or state type. For example, the trigger event, *Did-restrict anchoree (secure-anchoree) in pa (the-pa)* exemplifies the did-restrict event. For simplicity, these exemplifies links do not appear in the illustration. In addition, Append-to-Secure-Anchoree's action and result are control actions and events (do-focus-on and did-focus-on) whose parameters (prescription, goal, and rationale) are represented as conceptual networks comprising each focus component (e.g., prescription, goal, rationale), each component's constituent action, event, and state patterns, and appropiate links among them. These patterns also exemplify particular action, event, and state types.

Figure 39. An ACCORD KSAR. This Append-to-Secure-Anchoree KSAR is represented as a conceptual network comprising each knowledge source component, each component's constituent action, event, and state patterns, and appropriate links among them. Each of these patterns matches or instantiates the corresponding pattern in the Yoke-Structures knowledge source. For example, KSAR37's trigger event, *Did-anchor helix2-1 to helix1-1 in pa1 with NOE1* matches Append-to-Secure-Anchoree's trigger event, *Did-restrictanchoree (secure-anchoree) in pa (the-pa)* because did-anchor entails did-restrict, helix2-1 plays anchoree and pa1 plays pa. Again, for simplicity, these links do not appear in the illustration. In addition, KSAR37's action and result are control actions and events whose parameters are represented as conceptual networks whose constituent action, event, and state patterns instantiate the corresponding patterns in the Append-to-Secure-Anchorees knowledge source (see Figure 38). For example, *Do-append appendage to helix2-1 in pa1 with constraints* instantiates *Do-append appendage to secure-anchoree in the-pa with constraints* because helix2-1 is the value bound to secure-anchoree.

(next page)

Figure 39
(caption on preceding page)



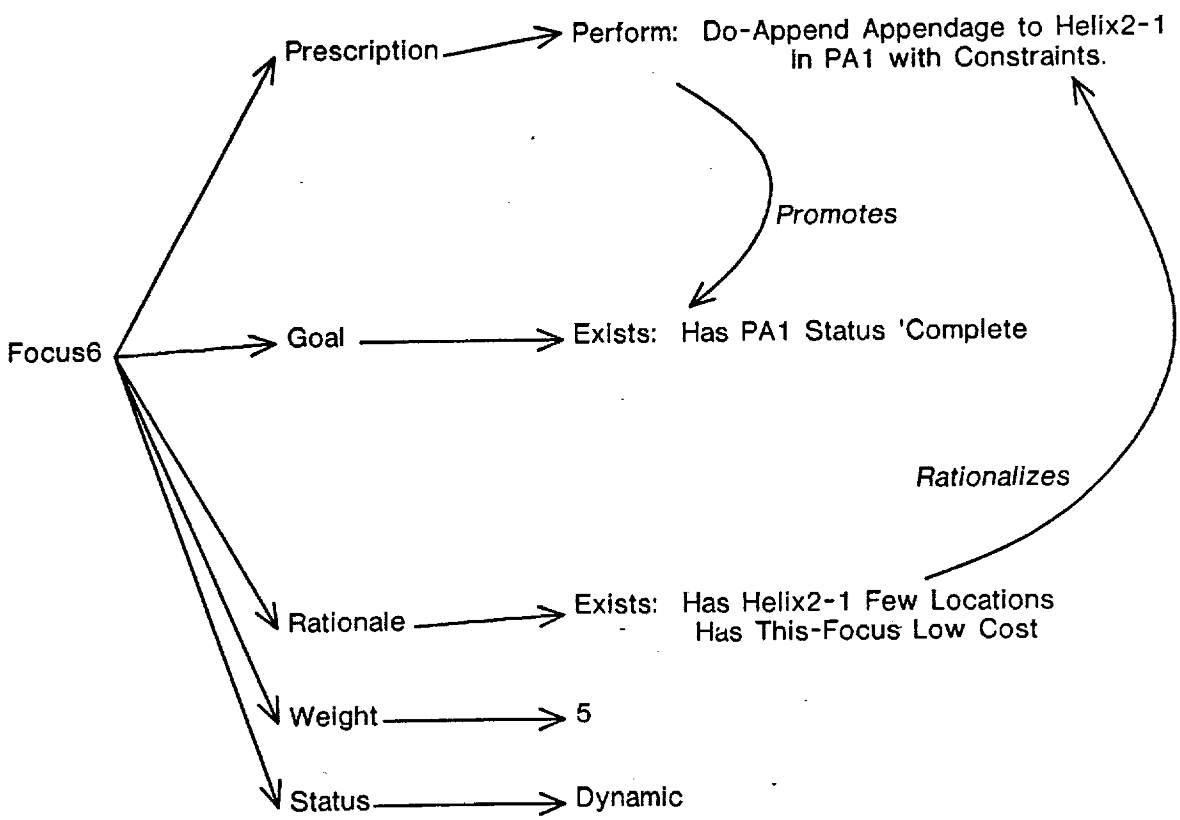


Figure 40. An ACCORD Focus Decision. Focus6 is represented as a conceptual network comprising each focus component (e.g., prescription, goal), each component's constituent action, event, and state patterns, and the links among them. Each pattern instantiates the corresponding pattern in the Append-to-Secure-Anchorees knowledge source. For example, *Do-append appendage to helix2-1 in pa1 with constraints* instantiates *Do-append appendage to secure-anchoree in the-pa with constraints* because helix2-1 is the value bound to secure-anchoree.

```
Strategy
    Perform: Develop PA With Best Anchor
       |----------------------------------------------------------------------->

Focus
    Perform: Do-Create Partial-Arrangement
    |----|
    Perform: Do-Include Secondary-Structure
         |-----------------|
    Perform: Do-Orient PA1 About Long Constraining Helix
                           |--------|
    Perform: Systematically Do-Position Rigid Long Constraining Constrained
       Secondary-Structure In PA1 with Strong Constraint
                                    |--------------------------------------->

Cycle |----------|-----------|-----------|------------|------------|------------|
      0          5          10          15           20           25           30
```

Figure 41. ACCORD Representation of a Simple PROTEAN Control Plan. This control plan is semantically equivalent to the plan shown in Figure 13. ACCORD's English-language representation captures related sets of heuristics in concise, perspicuous control sentences. While this figure displays only the prescriptions of control decisions, similar ACCORD provides similarly concise and perspicuous representations of their goals and rationales.

```
Strategy
     Perform: Quickly Do-Position Long Constraining Secondary-Structure in Current-Best
                PA with Strong Constraints
         |----------------------------------------------------------------------->

Sub-Strategy
     Perform: Quickly Do-Position Long Constraining Secondary-Structure (Target-Object)
                In PA1 with Strong Constraints
          |--------------------------------|
     Perform: Quickly Do-Position Long Constraining Secondary-Structure (Target-Object)
                In PA2 with Strong Constraints
                                           |------------------------------------>

Focus
     Perform: Quickly Do-Position Helix3-1 in PA1 with Strong Constraints
          |--------|
     Perform: Quickly Do-Position Helix4-1 in PA1 with Strong Constraints
                    |-------------------|
     Perform: Quickly Do-Position Helix4-2 in PA2 with Strong Constraints
                                           |---------------|
     Perform: Quickly Do-Position Helix6-2 in PA2 with Strong Constraints
                                                           |--------------->

Cycle |----------|-----------|-----------|------------|------------|------------|
      0          5           10          15           20           25           30
```

Figure 42. Excerpt from a More Complex PROTEAN Control Plan. In more complex control plans, ACCORD clearly articulates the hierarchical relationships between control decisions, whereby the higher-level decisions summarize and prescribe sequences of subordinate decisions. In this example, the generic control knowledge source, Refine-Parameters, generates the excerpted plan automatically. Starting with the top-level strategy, it substitutes the values pa1 and then pa2 for the phrase, current-best pa, to generate the sequence of two sub-strategies. For each sub-strategy, it similarly substitutes values best first for the phrase, long constraining secondary-structure, to generate sequences of focus decisions. Again, ACCORD provides similarly concise and perspicuous representations of the goals and rationales of all control decisions.

I recommend KSAR55:

Anchor Helix6-2 to Helix3-2 in PA2 with NOE7.

> **Why?**

I recommend KSAR55:

Anchor Helix6-2 to Helix3-2 in PA2 with NOE7.

because I am trying to:

F7: Quickly Do-Position Helix6-2 in PA2 with Strong Constraints.

> **Why NOE7?**

NOE7 is Strong (R60).
NOE7 is a Constraint (R100).

> **Why F7?**

I am attempting to:

F7: Quickly Do-Position Helix6-2 in PA2 with Strong Constraints.

because I am attempting to:

S3-2: Quickly Do-Position Long Constraining Secondary-Structure
in PA2 with Strong Constraints.

> **Why Helix6-2?**

Helix 6-2 is Long (80).
Helix 6-2 is Constraining (75).
Helix 6-2 is a Secondary-Structure (100).

> **Why S3-2?**

I am trying to:

S3-2: Quickly Do-Position Long Constraining Secondary-Structure
in PA2 with Strong Constraint.

because I am trying to:

S3: Quickly Do-Position Long Constraining Secondary-Structure
in Current-Best PA with Strong Constraint.

Figure 43. Explanation of the Decision to Perform KSAR55 expressed in ACCORD.

I recommend:
KSAR56: Yoke Helix6-2 with Helix4-2 in PA2 with NOE9.

> Override with KSAR55.

I am performing:
KSAR55: Anchor Helix6-2 to Helix3-2 in PA2 with NOE8.

. . .

I would like to understand why you prefer KSAR55 over KSAR56:

KSAR55: Anchor Helix6-2 to Helix3-2 in PA2 with NOE8.
KSAR56: Yoke Helix6-2 with Helix4-2 in PA2 with NOE9.

Is it because you prefer:
(a) Action = Do-Anchor over Action = Do-Yoke?
(b) Constraint = NOE9 over Constraint = NOE8?

> a

Do you prefer Action = Do-Anchor over Action = Do-Yoke
because you wish to:
Quickly Do-Position Helix6-2 in PA2 with Strong Constraint?

> Yes

In general, do you prefer to:
Quickly Do-Position Long Constraining Secondary-Structure
in Current-Best PA with Strong Constraint?

> Yes

I am modifying my current plan.
Shall I modify the knowledge source: Build-PAs?

> Yes

I am modifying the knowledge source: Build-PAs.

Figure 44. MARCK Learns to Prefer Anchoring Actions over Yoking Actions in the Context of ACCORD.

Figure 45. An Expert Arrangement-Assembler. The arrangement-assembler's conceptual network integrates PROTEAN's biochemistry knowledge and SIGHTPLAN's construction knowledge under ACCORD. Similarly, many knowledge sources shared by PROTEAN and SIGHTPLAN (not shown here) refer only to domain-independent entities (see Figures 36 and 38). With additional knowledge about refining prototypes, identifying similar problems, and assessing performance, the arrangement-assembler could automatically program new applications and transfer strategic knowledge among similar problems.

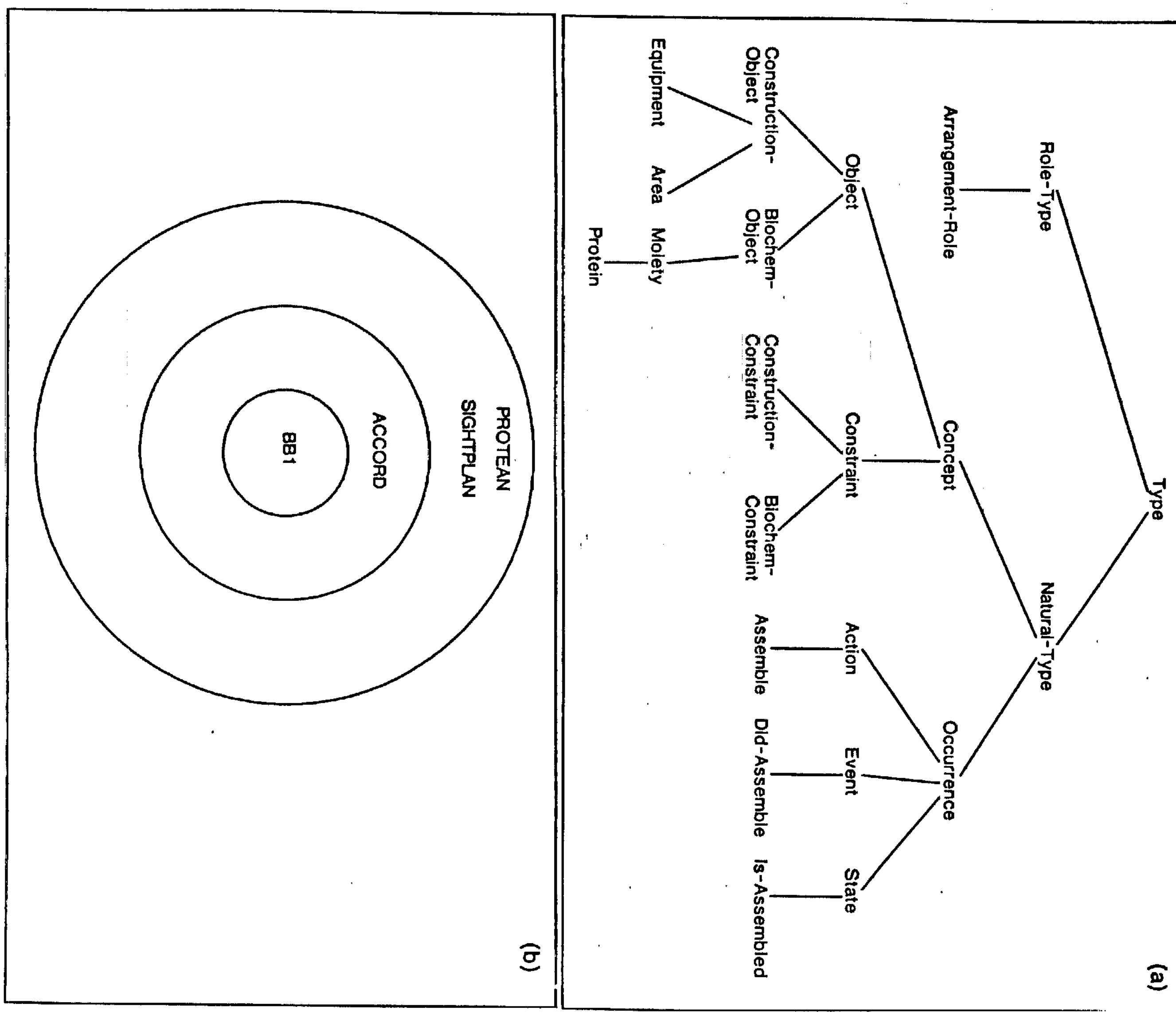
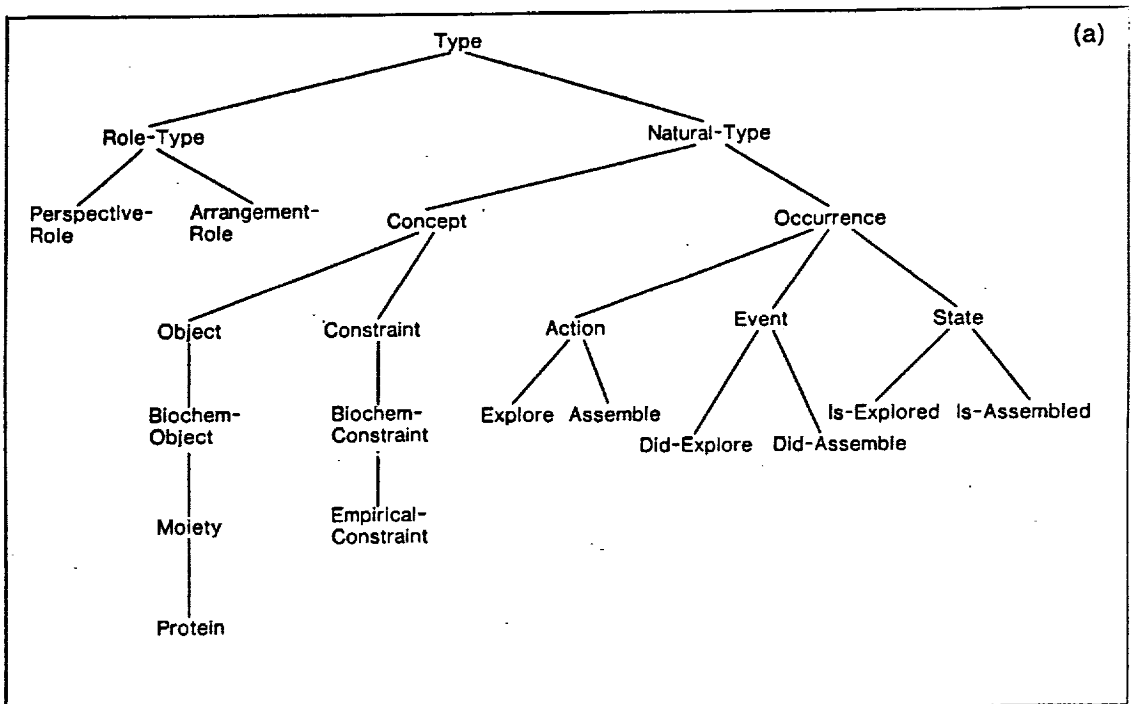


Figure 46. An Expert Protein-Analyzer. The protein-analyzer's conceptual network integrates ACCORD's knowledge of assembly actions, events, and states with EXPLORE's knowledge of exploration actions, events, and states and PROTEAN's and FEATURE's shared biochemistry knowledge. It incorporates all of PROTEAN's and FEATURE's knowledge sources (not shown here). With additional knowledge about combining its actions for particular purposes, the protein-analyzer could, for example: (a) solve a test protein and then examine it for interesting features; or (b) search for interesting while solving a protein and pursue only hypothesized conformations that exhibit interesting features.



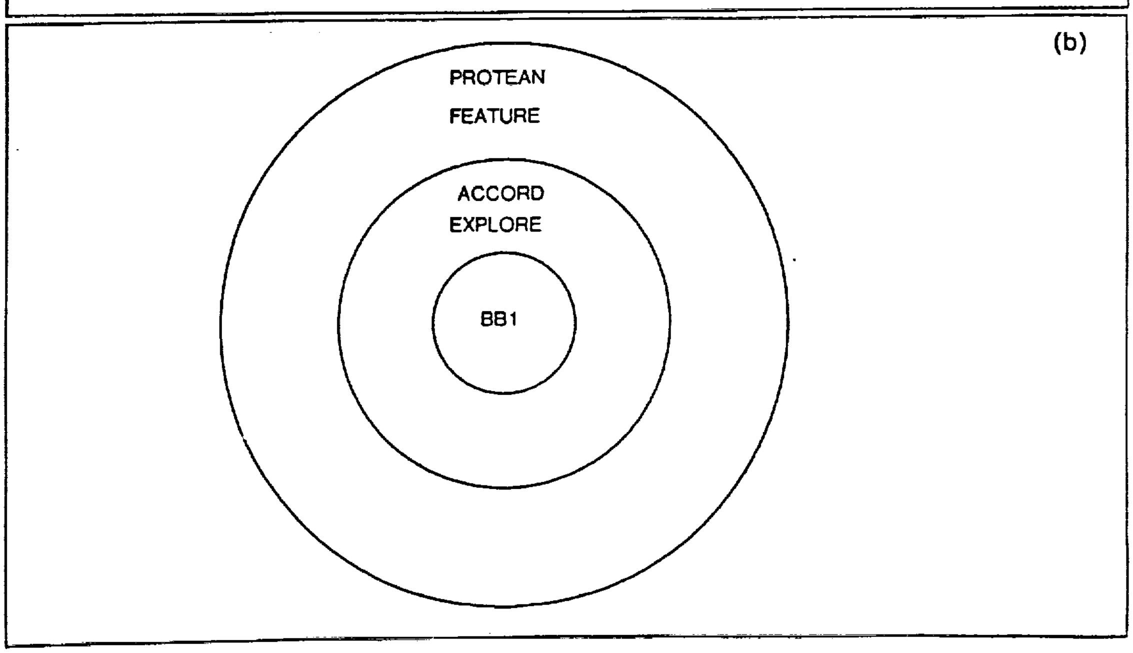


Table 1. Descriptions of Some BB1 Application Systems.

Architecture	Description
BB1[23]	Blackboard-based problem solving architecture
Framework	**Description**
ACCORD	Solves arrangement problems using the assembly method
EXPLORE[2]	Notices interesting features of proteins using perspectives
Application	**Description**
AVC[43]	Plans missions for automomous vehicles
FEATURE[3]	Explores protein structures for interesting features
ICP[34]	Dynamically plans curricula for an intelligent tutoring system
KRYPTO[28]	Solves constraint-satisfaction problems
PHRED[41]	Plans the construction process for aircraft components
PROCHEM[11]	Models protein structure based on theoretical constraints
PROTEAN[4,25,29]	Assembles protein structure based on empirical constraints
RAPS[30]	Diagnoses electro-mechanical systems
SADVISOR[10]	Advises on space station safety
SIGHTPLAN[50]	Designs construction site layouts
SIMLAB[40]	Schedules personnel, hardware and software for flight simulation

Table 2. Some of the Constraints Available to PROTEAN

Primary structure
Atomic structure of individual amino acids
Van der Waals' radii of individual atoms
Peptide bond geometry
Secondary structure
Architectures of alpha-helices and beta-sheets
Molecular size
Molecular shape
Molecular density
NOE measurements
Surface data

Table 3. Methods for Solving Arrangement Problems.

---

1. **Select** an arrangement that satisfies the constraints from a pre-enumerated set of alternatives.
   Requires Knowledge of: Alternative arrangements.
   Example: A travel agent selects one of several tour "packages" that includes all of the destinations requested by a client.

2. **Refine** a prototypical arrangement so as to satisfy the constraints.
   Requires Knowledge of: A prototypical arrangement.
   Example: An architect refines a prototypical U-shaped kitchen design to include the special appliances requested by a client.

3. **Modify** an almost-correct arrangement to satisfy the constraints.
   Requires Knowledge of: Almost-correct arrangements.
   Example: A tool designer modifies an existing tool to fit a new machine.

4. **Generate** a complete arrangement that satisfies the constraints.
   Requires Knowledge of: A procedure for generating complete arrangements.
   Example: A psychologist uses a multi-dimensional-scaling algorithm to generate a spatial model of subjects' similarity ratings of related concepts.

5. **Construct** an arrangement that satisfies the constraints.
   Requires Knowledge of: A method for constructing arrangements.
   Example: A person solves a jigsaw puzzle by placing pieces one at a time.

Table 4. Templates for Arrangement-Assembly Root Verbs.

- Assemble pa
  - Define pa
    - Create pa at level
    - Include object in pa
    - Orient pa about included-object
  - Position object in pa with constraints
    - Anchor anchoree to anchor in pa with constraints
    - Restrict included-object in pa with constraints
    - Yoke included-object to included-object in pa with constraints
    - Append appendage to included-object in pa with constraints
    - Consolidate included-objects in pa with constraints
  - Integrate pa with pa
    - Merge pa with pa
    - Incorporate pa into pa via included-object
    - Dock pa to pa with constraints
  - Coordinate pa at level and level
    - Refine sub-object of object in pa from level to level
    - Adjust object for sub-object in pa

Table 5. Examples of ACCORD Prescriptions

1. Perform an action in a particular class of actions.
   Perform: Do-Position Long Helix in PA1 with Strong Constraint.

2. Perform an action that was triggered by a particular class of events.
   Respond-to-Events-that: Did-Restrict Well-Restricted Anchoree in PA1 with Constraint.

3. Perform an action that was enabled by a particular class of states.
   Respond-to-States-in-which: Has Anchoree Few Locations.

4. Perform an action that causes a particular class of events.
   Cause: Did-Restrict Helix2-1 in PA1 with Constraint.

5. Perform an action that promotes a particular class of states.
   Perform: Is-Positioned Helix2-1 in PA1 with Strong Constraint.

Table 6. Examples of ACCORD Goals.

1. Achieve a state in which a particular class of events has occurred.
   Until:
      Did-Restrict Helix2-1 in PA1 with Constraint.

2. Achieve a state in which a particular class of actions is executable.
   Until:
      Can Perform:
         Do-Append Helix2-3 to Helix in PA1 with Constraint.

3. Achieve a state in which a particular class of actions has been executed.
   Did Perform:
      Do-Append Helix2-3 to Helix in PA1 with Constraint.